ANÁLISIS DEL RETO 4

Alejandro Pardo Sánchez, 202223709, a.pardos2@uniandes.edu.co

Joseph Eli Pulido Gómez, 202211365, je.pulidog1@uniandes.edu.co

Santiago González Serna, 202021226, s.gonzalezs@uniandes.edu.co

Carga de datos

Descripción

```python
def new_data_structs():
    """
    Inicializa las estructuras de datos del modelo. Las crea de
    manera vacía para posteriormente almacenar la información.
    """
    #TODO: Inicializar las estructuras de datos
    data_structs = {"dirigido": None,
                    "no_dirigido":None,
                    "mp_lobos":None,
                    "lt_lobos_a2" :None,
                    "lt_todos": None,
                    "mp_puntos":None}

    data_structs["dirigido"] = gr.newGraph(datastructure='ADJ_LIST',
                                           directed=True,
                                           size=14000)
    data_structs["no_dirigido"] = gr.newGraph(datastructure='ADJ_LIST',
                                           directed=False,
                                           size=14000)
    data_structs["mp_lobos"] = mp.newMap(20,
                                maptype="PROBING",
                                loadfactor=0.5)
    data_structs["mp_lobos_a2"] = mp.newMap(20,
                                maptype="PROBING",
                                loadfactor=0.5)

    data_structs["lt_lobos_a2"] = lt.newList("ARRAY_LIST")
    data_structs["lt_todos"] = lt.newList("ARRAY_LIST")
    data_structs["mp_puntos"] =mp.newMap(20,
                                    maptype="PROBING",
                                    loadfactor=0.5)
    data_structs["puntos_encuentro"] = lt.newList("ARRAY_LIST")
    data_structs["Fechas"] = om.newMap(omaptype="RBT", cmpfunction=compareFecha)
    return data_structs
```

Entrada	- El data_structs, que contiene 9 llaves: un grafo dirigido, un grafo no dirigido, una tabla de hash con los id de los lobos, una tabla de hash con los id de los lobos para el archivo pequeño, una lista para el archivo pequeño, una lista para almacenar los datos del archivo grande, una lista con todos los puntos de encuentro y un árbol ordenado por fecha (timestamp)
Salidas	Un diccionario
Implementado (Sí/No)	Sí

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).

Procesadores	**11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz 2.42 GHz**
Memoria RAM	8 GB
Sistema Operativo	Windows 10

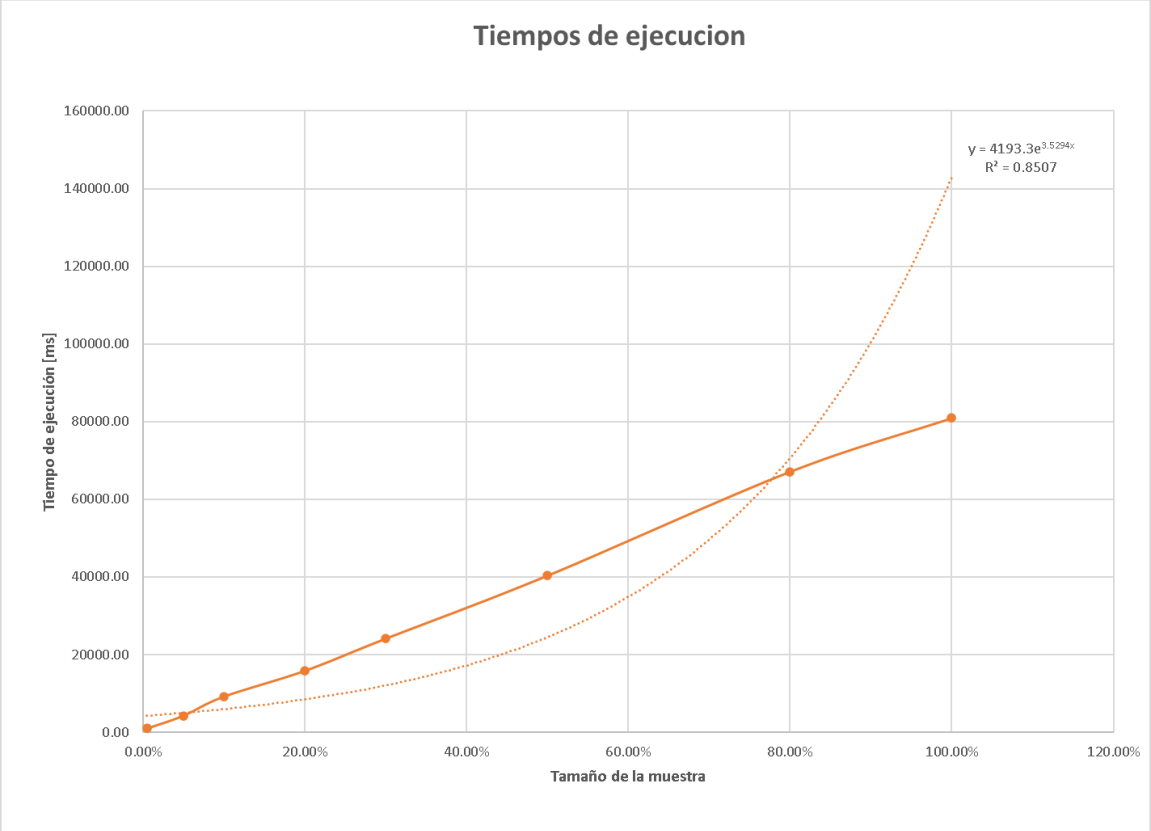
Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Porcentaje de la muestra [pct]	Tamaño de la muestra	Tiempo [ms]
0.50%	2391.00	982.73
5.00%	11959.00	4271.50
10.00%	23919.00	9204.90
20.00%	47838.00	15836.40
30.00%	71758.00	24114.30
50.00%	119597.00	40349.38
80.00%	191355.00	67114.90
100.00%	239194.00	80947.95

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.

La línea de tendencia de la complejidad es casi lineal, por lo que se infiere una aceptable complejidad temporal.

Requerimiento 1

Descripción

```python
def req_1(data_structs, vertexA, vertexB):
    """
    Función que soluciona el requerimiento 1
    """
    # TODO: Realizar el requerimiento 1
    if not gr.containsVertex(data_structs["dirigido"], vertexA):
        return False
    if not gr.containsVertex(data_structs["dirigido"], vertexB):
        return False
    search = dfs.DepthFirstSearch(data_structs["dirigido"], vertexA)
    camino  = dfs.pathTo(search, vertexB)
    existe = str(dfs.hasPathTo(search, vertexB))
    tama_cam = st.size(camino)
    punto_encuentro = 0
    vertices_seg = 0
    lista_distancia = lt.newList()
    for dato in lt.iterator(camino):
        lt.addLast(lista_distancia, dato)
        tama = dato.split("_")
        if len(tama) == 2:
            punto_encuentro +=1
        else:
            vertices_seg +=1
    distancia = 0
    i = 0
    while i < lt.size(lista_distancia):
        act = lt.getElement(lista_distancia, i)
        sig = lt.getElement(lista_distancia, i+1)
        act_lon = inverso_coo(act)[0]
        act_lat = inverso_coo(act)[1]
        sig_lon = inverso_coo(sig)[0]
        sig_lat = inverso_coo(sig)[1]
        distancia += harvesine_simple(act_lon, act_lat, sig_lon, sig_lat)
        i+=1

    tabular = tabularR1_R2(data_structs, camino)
    return existe, tama_cam, punto_encuentro, vertices_seg, tabular, round(distancia,6)
```

Entrada	- El data_structs - Identificador del punto de encuentro de origen - Identificador del punto de encuentro de destino
Salidas	Una tupla de 6 elementos, si existe el vértice de destino en el árbol dfs, el total de nodos en el camino, el total de puntos de encuentro en el camino, el total de puntos de seguimiento en el camino, una lista con los datos para ser tabulados y la distancia total del camino.
Implementado (Sí/No)	Sí

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

K < n, debido a que no se recorren todos los datos sino solo los del intervalo especifico.

Pasos	**Complejidad**
Gr.contains	O(V)
DFS	O(V+E)
For	O(k)
addLast	O(1)
While	O(k)
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).

Procesadores	**11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz 2.42 GHz**
Memoria RAM	8 GB
Sistema Operativo	Windows 10

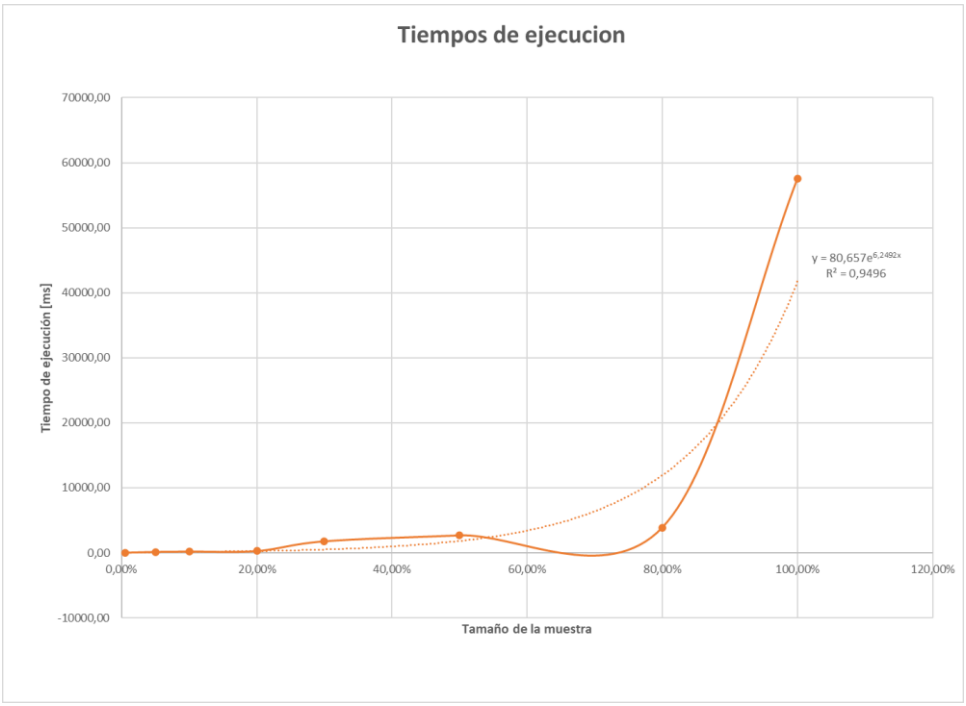
Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Porcentaje de la muestra [pct]	Tamaño de la muestra	Tiempo [ms]
0,50%	2391.00	23.69
5,00%	11959.00	138.29
10,00%	23919.00	198.85
20,00%	47838.00	271.09
30,00%	71758.00	1770.89
50,00%	119597.00	2685.46
80,00%	191355.00	3878.80
100,00%	239194.00	57553.15

Graficas

Las gráficas con la representación de las pruebas realizadas.


Tiempos de ejecucion
70000,00
60000,00
50000,00
40000,00
30000,00
20000,00
10000,00
0,00
-10000,00
Tiempo de ejecución [ms]
0,00%
20,00%
40,00%
60,00%
80,00%
100,00%
120,00%
Tamaño de la muestra
y = 80,657e6,2492x
R² = 0,9496

Análisis

Podemos ver que la compejidad se ajusta con la linealidad de las misma. En este algoritmo se utilizó un algoritmo DFS en el cual se recorrido el grafo según sus adyacentes para asi encontrar un camino para encontrar el camino solicitado por el usuario.

Requerimiento 2

Descripción

```
def req_2(data_structs, vertex_i, vertex_f ):
    """
    Función que soluciona el requerimiento 2
    """
    # TODO: Realizar el requerimiento 2
    if not gr.containsVertex(data_structs["dirigido"], vertex_i):
        return False
    if not gr.containsVertex(data_structs["dirigido"], vertex_f):
        return False
    find = bfs.BreadhtFisrtSearch(data_structs["dirigido"], vertex_i)
    path = bfs.pathTo(find, vertex_f)
    existe = str(bfs.hasPathTo(find,vertex_f))
    puntos_encuentro = 0
    vertices_seguidos = 0
    lista_recorrido = lt.newList()
    for dato in lt.iterator(path):
        lt.addLast(lista_recorrido, dato)
        nodo = dato.split("_")
        if len(nodo) == 2:
            puntos_encuentro += 1
        else:
            vertices_seguidos +=1
    nodes = lt.size(lista_recorrido)
    distancia = 0
    i =0
    while i < lt.size(lista_recorrido):
        act = lt.getElement(lista_recorrido, i)
        sig = lt.getElement(lista_recorrido, i+1)
        act = (inverso_coo(act)[0],inverso_coo(act)[1])
        sig = (inverso_coo(sig)[0], inverso_coo(sig)[1])
        distancia += harvesine_simple(act[0], act[1], sig[0], sig[1])
        i+=1
    tabla = tabular(data_structs, path)
    return existe, nodes, puntos_encuentro, vertices_seguidos, tabla, round(distancia,6)
```

Entrada	- El data_structs - Identificador del punto de encuentro de origen - Identificador del punto de encuentro de destino
Salidas	Una tupla de 6 elementos, si existe el vértice de destino en el árbol bfs, el total de nodos en el camino, el total de puntos de encuentro en el camino, el total de puntos de seguimiento en el camino, una lista con los datos para ser tabulados y la distancia total del camino.
Implementado (Sí/No)	Sí

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

K < n, debido a que no se recorren todos los datos sino solo los del intervalo especifico.

Pasos	Complejidad
Creacion Array_list	O(1)
for	O(n)
BFS	O((V+E))
Condicionales	O(n)
Funciones auxiliares	O(n)
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).

Procesadores	**11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz 2.42 GHz**
Memoria RAM	8 GB
Sistema Operativo	Windows 10

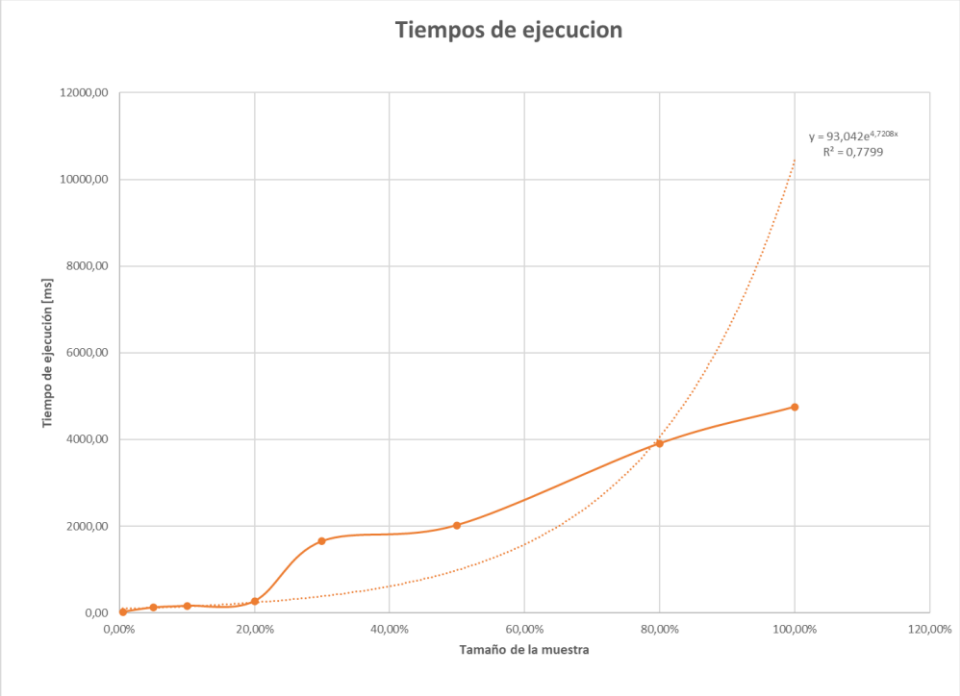
Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Porcentaje de la muestra [pct]	Tamaño de la muestra	Tiempo [ms]
0,50%	2391,00	20,35
5,00%	11959,00	120,04
10,00%	23919,00	159,27
20,00%	47838,00	266,39
30,00%	71758,00	1650,43
50,00%	119597,00	2021,40
80,00%	191355,00	3908,38
100,00%	239194,00	4753,53

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

La grafica tiene un comportamiento casi lineal ascendente, esto quiere decir que su complejidad temporal es muy cercana a la hallada.

En este algoritmo se utilizó el algoritmo BFS para encontrar el camino, también se puedo instaurar el algoritmo DFS, pero en este caso no hay ventajas ni desventajas si se implementa o no.

Requerimiento 3

Descripción

```
def req_3(data_structs):
    """
    Función que soluciona el requerimiento 3
    """
    # TODO: Realizar el requerimiento 3
    kosaraju = scc.KosarajuSCC(data_structs["dirigido"])
    conectados = scc.connectedComponents(kosaraju)
    mapa_ids = kosaraju["idscc"]
    mapa_nuevo = mp.newMap(20, maptype="PROBING", loadfactor=0.5)
    for llaves in lt.iterator(mp.keySet(mapa_ids)):
        valores = me.getValue(mp.get(mapa_ids, llaves))
        agregar_id(mapa_nuevo, valores, llaves)
    lt_ordenar_mayor = lt.newList()
    for keys in lt.iterator(mp.keySet(mapa_nuevo)):
        lista = me.getValue(mp.get(mapa_nuevo, keys))
        lt.addLast(lt_ordenar_mayor, lt.size(lista))
    merg.sort(lt_ordenar_mayor, ordenar_mayor)
    top = lt.newList()
    a=1
    for tamano in lt.iterator(lt_ordenar_mayor):
        for llave in lt.iterator(mp.keySet(mapa_nuevo)):
            lista = me.getValue(mp.get(mapa_nuevo, llave))
            if lt.size(lista) == tamano:
                anadir = {"IDSCC":llave, "NODE_IDS":lista}
                lt.addLast(top, anadir)
                break
        a+=1
        if a == 6:
            break
    tabular = tabla_req3(data_structs, top)
    return conectados, tabular
```

Entrada	- El data_structs, el requerimiento se ejecuta sobre todo el grafo
Salidas	Una tupla de 2 elementos, la cantidad de elementos fuertemente conectados en el grafo dirigido y una lista para tabular la información.
Implementado (Sí/No)	Sí (Alejandro Pardo)

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

K < n, debido a que no se recorren todos los datos sino solo los de la clase especifica.

Pasos	**Complejidad**
Kosaraju	O(V*E)
For	O(V)
For anidado	O(k^2)
AddLast (todos)	O(1)
Merge sort (todos)	O(k*log(k))
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).

Procesadores	**11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz 2.42 GHz**
Memoria RAM	8 GB
Sistema Operativo	Windows 10

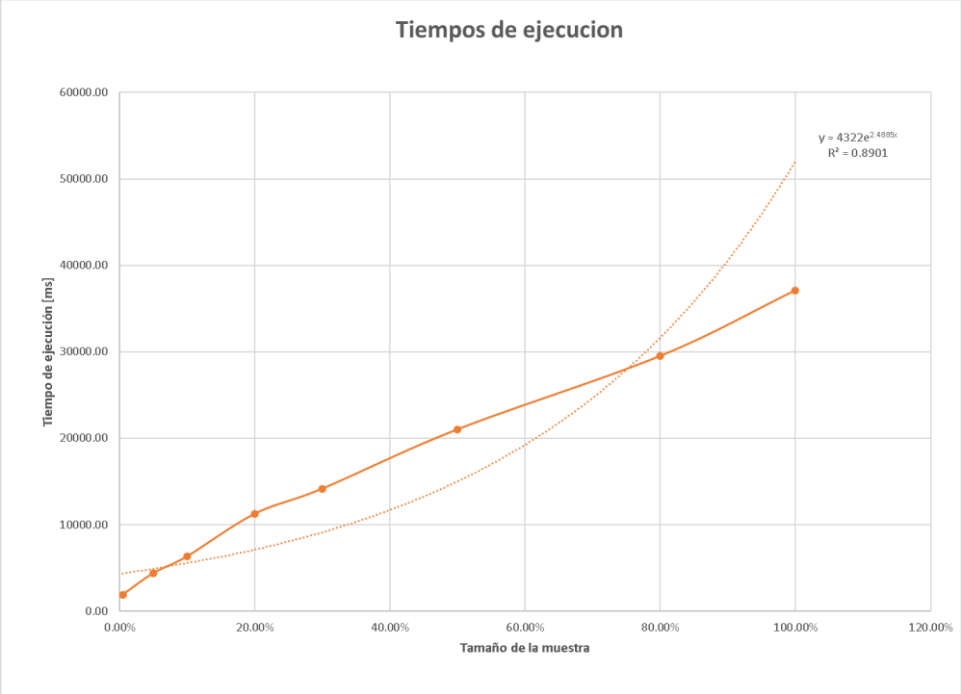
Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Porcentaje de la muestra [pct]	Tamaño de la muestra	Tiempo [ms]
0.50%	2391.00	1890.97
5.00%	11959.00	4362.93
10.00%	23919.00	6310.14
20.00%	47838.00	11230.73
30.00%	71758.00	14143.88
50.00%	119597.00	21005.49
80.00%	191355.00	29508.74
100.00%	239194.00	37098.54

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

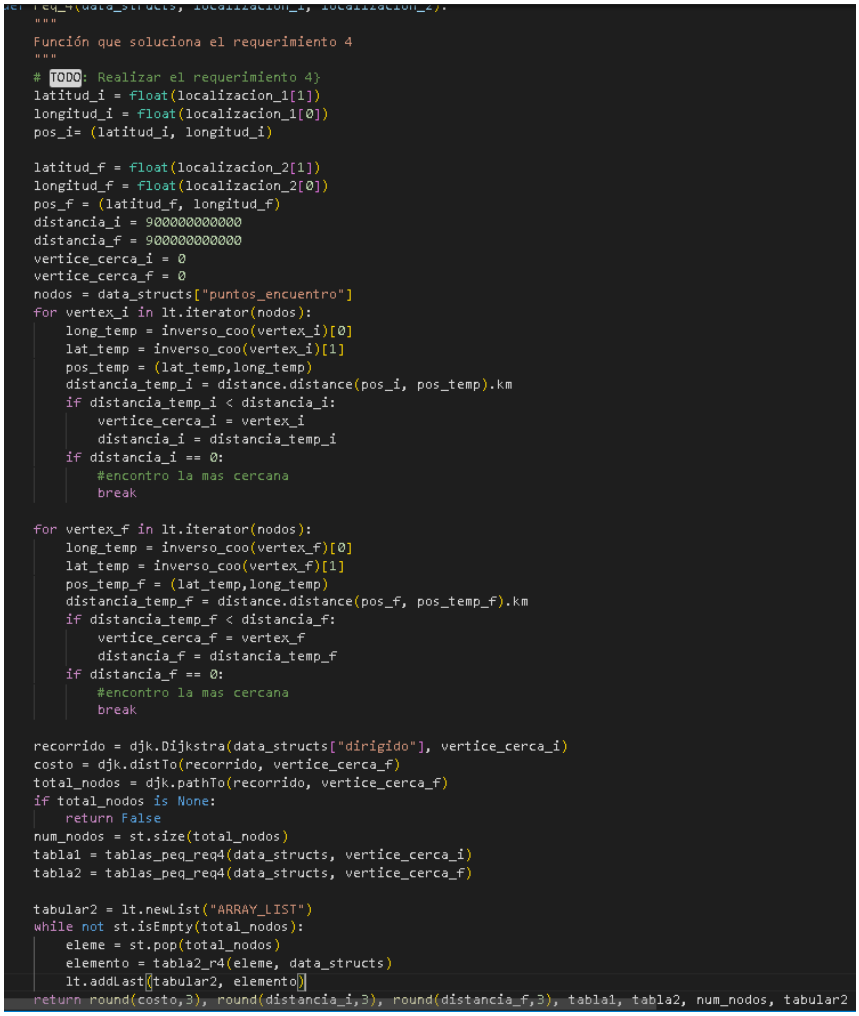
Análisis de resultados de la implementación, tener cuenta las pruebas realizadas y el analisis de complejidad.

Se utilizo el algoritmo Kosaraju para encontrar los componentes fuertemente conectados, estos serían las manadas que existen en el grupo de lobos.

La línea de tendencia muestra un crecimiento constante, casi lineal donde cada vez la pendiente aumenta, puede relacionarse a la complejidad planteada: O(V*E), por lo tanto, es posible inferir parcialmente que la complejidad que se calculó corresponde con la complejidad evidenciada en la gráfica.

Requerimiento 4

Descripción



```
    """
    Función que soluciona el requerimiento 4
    """
    # TODO: Realizar el requerimiento 4}
    latitud_i = float(localizacion_1[1])
    longitud_i = float(localizacion_1[0])
    pos_i= (latitud_i, longitud_i)

    latitud_f = float(localizacion_2[1])
    longitud_f = float(localizacion_2[0])
    pos_f = (latitud_f, longitud_f)
    distancia_i = 900000000000
    distancia_f = 900000000000
    vertice_cerca_i = 0
    vertice_cerca_f = 0
    nodos = data_structs["puntos_encuentro"]
    for vertex_i in lt.iterator(nodos):
        long_temp = inverso_coo(vertex_i)[0]
        lat_temp = inverso_coo(vertex_i)[1]
        pos_temp = (lat_temp,long_temp)
        distancia_temp_i = distance.distance(pos_i, pos_temp).km
        if distancia_temp_i < distancia_i:
            vertice_cerca_i = vertex_i
            distancia_i = distancia_temp_i
        if distancia_i == 0:
            #encontro la mas cercana
            break

    for vertex_f in lt.iterator(nodos):
        long_temp = inverso_coo(vertex_f)[0]
        lat_temp = inverso_coo(vertex_f)[1]
        pos_temp_f = (lat_temp,long_temp)
        distancia_temp_f = distance.distance(pos_f, pos_temp_f).km
        if distancia_temp_f < distancia_f:
            vertice_cerca_f = vertex_f
            distancia_f = distancia_temp_f
        if distancia_f == 0:
            #encontro la mas cercana
            break

    recorrido = djk.Dijkstra(data_structs["dirigido"], vertice_cerca_i)
    costo = djk.distTo(recorrido, vertice_cerca_f)
    total_nodos = djk.pathTo(recorrido, vertice_cerca_f)
    if total_nodos is None:
        return False
    num_nodos = st.size(total_nodos)
    tabla1 = tablas_peq_req4(data_structs, vertice_cerca_i)
    tabla2 = tablas_peq_req4(data_structs, vertice_cerca_f)

    tabular2 = lt.newList("ARRAY_LIST")
    while not st.isEmpty(total_nodos):
        eleme = st.pop(total_nodos)
        elemento = tabla2_r4(eleme, data_structs)
        lt.addLast(tabular2, elemento)
    return round(costo,3), round(distancia_i,3), round(distancia_f,3), tabla1, tabla2, num_nodos, tabular2
```

Entrada	- El data_structs - Localización geográfica del punto de origen (longitud y latitud). -Localización geográfica del punto de destino (longitud y latitud).
Salidas	Una tupla de 7 elementos, la distancia total del recorrido, la distancia entre la coordenada 1 hasta el nodo más cercano, la distancia entre la coordenada 2 hasta el nodo más cercano, una lista con la información del nodo más cercano a la localización 1 para ser tabulada, una lista con la información del nodo más cercano a la localización 2 para ser tabulada, el total de nodos visitados y una lista con los datos del camino para ser tabulada.
Implementado (Sí/No)	Sí (Joseph Pulido)

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Para un k < n donde n es el número total de datos. Esto es porque no se recorren todos los datos, solamente el tramo dado por parámetro

Pasos	**Complejidad**
Creacion Array_list	O(1)
Funciones auxiliadoras	O(k)
for	O(k) k numero de puntos de encuentro
Comparaciones	O(k)
addLast	O(1)
Algoritmo Dijkstra	O((E)*log(V))
TOTAL	***O((E)*log(V))***

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).

Procesadores	**11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz 2.42 GHz**
Memoria RAM	8 GB
Sistema Operativo	Windows 10

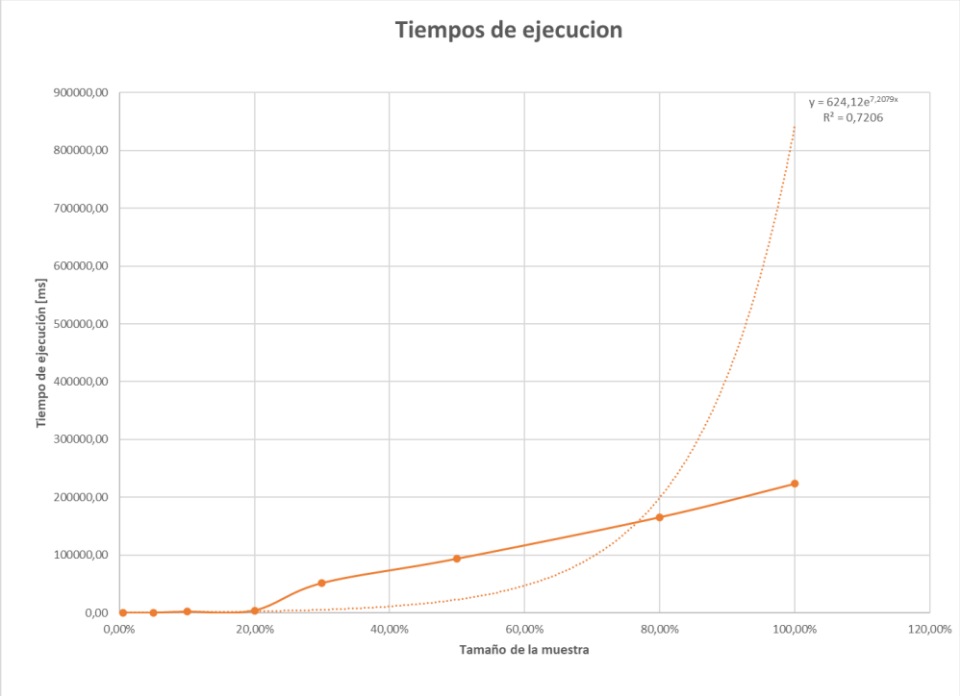
Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Porcentaje de la muestra [pct]	**Tamaño de la muestra**	**Tiempo [ms]**
0,50%	2391,00	58,99
5,00%	11959,00	431,82
10,00%	23919,00	2150,08
20,00%	47838,00	4169,64
30,00%	71758,00	51628,22
50,00%	119597,00	93850,88
80,00%	191355,00	165617,72
100,00%	239194,00	223518,80

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

Podemos ver en la gráfica que la linea de tendencia se comporta como la complejidad evaluada, lo que quiere decir que hay una precisión en cuanto la complejidad calculada

En este requerimiento se empleó el algoritmo Dijkstra debido a que es más rápido que algoritmos como Bellmanford, Kruskal, etc. Debido a que la complejidad temporal era la que mas se adaptaba mejor al problema ya que era un grafo muy extenso y con ciertas características que al momento de tratar de implementar los demás algoritmos daba algún error.

Requerimiento 5

```python
def req_5(data_structs, origen, distancia_max, puntos_min):
    """
    Función que soluciona el requerimiento 5
    """
    #DATOS INICIALES
    grafo = data_structs["no_dirigido"]
    recorrido_0 = lt.newList("ARRAY_LIST")
    lt.addLast(recorrido_0, origen)
    dato_0 = (0, recorrido_0)
    distancia = distancia_max/2
    caminos = lt.newList("ARRAY_LIST")
    cola_distancias = mpq.newMinPQ(cmp_menor_distancia)
    mpq.insert(cola_distancias, dato_0)

    while mpq.isEmpty(cola_distancias) == False:
        dato = mpq.delMin(cola_distancias)
        distancia_act = dato[0]
        recorrido_act = dato[1]
        nodo_act = lt.lastElement(recorrido_act)

        #OBTENER LONGITUD Y LATITUD
        nodo_act = nodo_act.replace("m", "-")
        nodo_act = nodo_act.replace("p", ".")
        spl = nodo_act.split("_")
        lon_act = float(spl[0])
        lat_act = float(spl[1])

        if (distancia_act <= distancia) and (len(recorrido_act) >= puntos_min):
            lt.addLast(caminos, dato)
```

```python
        if (distancia_act <= distancia) and (len(recorrido_act) >= puntos_min):
            lt.addLast(caminos, dato)

        if (distancia_act <= distancia) and (len(recorrido_act) < puntos_min):
            for v in gr.adjacents(grafo, nodo_act):
                #OBTENER LONGITUD Y LATITUD
                v = v.replace("m", "-")
                v = v.replace("p", ".")
                spl = v.split("_")
                lon = float(spl[0])
                lat = float(spl[1])
                #NUEVOS DATOS
                distancia_new = distancia_act + harvesine_simple(lon_act, lat_act, lon, lat)
                recorrido_new = lt.addLast(v)
                dato_new = (distancia_new, recorrido_new)
                mpq.insert(dato_new)

    merg.sort(caminos, cmp_mas_puntos)

    #RESULTADOS
    num_max = lt.size(caminos) #1
    corredor_max = lt.firstElement(caminos) #2
    secuencia = corredor_max[1] #2A
    puntos_enc = lt.size(secuencia) #2B
    distancia_rec = corredor_max[0] #2C
    #2D: La secuencia del número posible de individuos visibles en el trayecto
    return num_max, corredor_max, secuencia, puntos_enc, distancia_rec
```

Descripción

Entrada	-Data structs: contiene los grafos con los datos utilizados -Origen: nodo de partida -Distancia_max: distancia máxima a recorrer -Puntos_min: número mínimo de nodos a visitar
Salidas	Una lista de DISClib con los caminos que cumplen con las condiciones (comenzar por el origen, recorrer como máximo distancia_max y pasar por mínimo puntos_min); el recorrido con más puntos, el número de puntos y la distancia recorrida.
Implementado (Sí/No)	Sí (Santiago González)

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Para un k < n donde n es el número total de datos. Esto es porque no se recorren todos los datos, solamente el tramo dado por parámetro

Pasos	**Complejidad**
Creacion Array_list	O(1)
Gr.adjacents	O(k)
For	O(k)
Comparacion	O(n)
AddLast	O(1)
Merge sort	O(k*log(k))
Dijkstra	O((E)*log(V))
TOTAL	***O((E)*log(V)***

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).

Procesadores	**11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz 2.42 GHz**
Memoria RAM	8 GB
Sistema Operativo	Windows 10

Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

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.

Requerimiento 6

Descripción

```
def req_6(data_structs, fecha1, fecha2, sexo):
    """
    Función que soluciona el requerimiento 6
    """
    # TODO: Realizar el requerimiento 6
    lista_intervalo = om.values(data_structs["Fechas"], fecha1, fecha2)

    mapa_nuevo = mp.newMap(20,
                           maptype="PROBING",
                           loadFactor=0.5)
    for cada_fecha in lt.iterator(lista_intervalo):
        for evento in lt.iterator(cada_fecha["Datos_fecha"]):
            id_lobo = evento["individual-local-identifier"]
            id_tag = evento["tag-local-identifier"]
            id_vertice = id_lobo+"_"+id_tag
            cada_lobo = lt.firstElement(me.getValue(mp.get(data_structs["mp_lobos_a2"], id_vertice)))
            if cada_lobo["animal-sex"] == sexo:
                existelobo = mp.contains(mapa_nuevo, id_vertice)
                if existelobo:
                    entry = mp.get(mapa_nuevo, id_vertice)
                    lobo_id = me.getValue(entry)
                    lon2 = float(lt.lastElement(lobo_id["Datos"])["location-long"])
                    lat2 = float(lt.lastElement(lobo_id["Datos"])["location-lat"])
                    lon1 = float(evento["location-long"])
                    lat1 = float(evento["location-lat"])
                    lobo_id["Distancia"] += harvesine_simple(lon1, lat1, lon2, lat2)
                    lt.addLast(lobo_id["Nodos"], evento)
                else:
                    lobo_id = {"Datos":lt.newList(),
                               "Distancia": 0,
                               "Nodos":lt.newList()}
                    mp.put(mapa_nuevo, id_vertice, lobo_id)
                lt.addLast(lobo_id["Datos"], evento)

    mas_distancia = 0
    menos_distancia = 5000000
    id_menos = ""
    id_mas = ""
    mas_nodo = ""
    menos_nodo = ""
    lista_nodos_mas = ""
    lista_nodos_menos = ""
    llaves_nuevo = mp.keySet(mapa_nuevo)
```

```
llaves_nuevo = mp.keySet(mapa_nuevo)
for cada_llave in lt.iterator(llaves_nuevo):
    lista_data_dis = me.getValue(mp.get(mapa_nuevo, cada_llave))
    dist = lista_data_dis["Distancia"]
    cant_nodos = lt.size(lista_data_dis["Nodos"])
    if dist > mas_distancia:
        mas_distancia = dist
        id_mas = cada_llave
        mas_nodo = cant_nodos
        lista_nodos_mas = lista_data_dis["Nodos"]
    if dist < menos_distancia:
        menos_distancia = dist
        id_menos = cada_llave
        menos_nodo = cant_nodos
        lista_nodos_menos = lista_data_dis["Nodos"]

mas = tabla_peq6(data_structs, id_mas, sexo, mas_distancia)
menos = tabla_peq6(data_structs, id_menos, sexo, menos_distancia)
menos_nodos = lt.size(lista_nodos_menos)
mas_nodos = lt.size(lista_nodos_mas)
tabla_menos = auxiliar_r6(lista_nodos_menos, data_structs)
tabla_mas = auxiliar_r6(lista_nodos_mas, data_structs)
return mas, menos, round(mas_distancia,3), round(menos_distancia,3), tabla_menos, menos_nodos, tabla_mas, mas_nodos
```

Entrada	- El data_structs - Fecha inicial del análisis - Fecha final del análisis - El sexo registrado del animal
Salidas	Una tupla de 8 elementos, una lista con la información del lobo que más recorrió en el intervalo de fechas, una lista con la información del lobo que menos recorrió en el intervalo de fechas, la distancia total del lobo que más recorrió, la distancia

	total del lobo que menos recorrió, una lista con la información del camino del lobo que menos recorrió para ser tabulada, el total de nodos que paso el lobo que menos recorrió, una lista con la información del camino del lobo que menos recorrió para ser tabulada y el total de nodos que paso el lobo que más recorrió.
Implementado (Sí/No)	Si

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Para un k < n donde n es el número total de datos. Esto es porque no se recorren todos los datos, solamente el tramo dado por parámetro

Pasos	**Complejidad**
Creacion Array_list	O(1)
Condicionales	O(1)
For	O(k)
For anidado	O(k^2)
Comparacion (todos)	O(k)
AddLast (todos)	O(1)
TOTAL	***O(k^2)***

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).

Procesadores	**11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz 2.42 GHz**
Memoria RAM	8 GB
Sistema Operativo	Windows 10

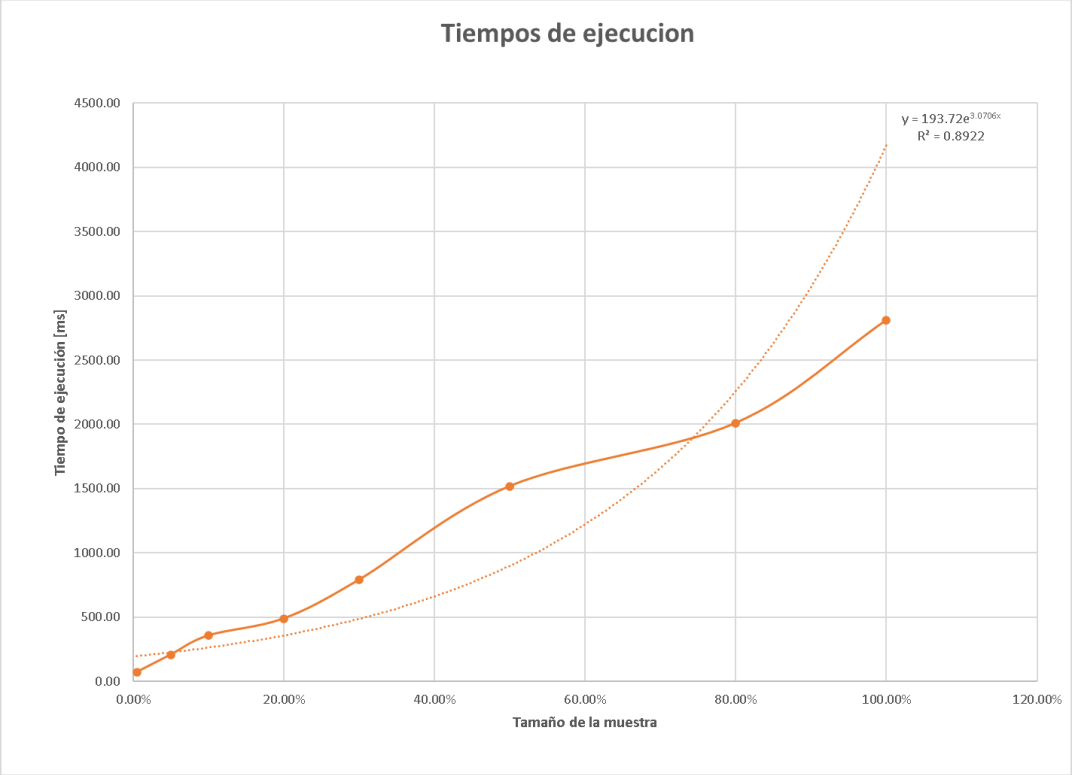
Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Porcentaje de la muestra [pct]	Tamaño de la muestra	Tiempo [ms]
0.50%	2391.00	71.17
5.00%	11959.00	206.84
10.00%	23919.00	355.58
20.00%	47838.00	488.20
30.00%	71758.00	789.38
50.00%	119597.00	1518.37
80.00%	191355.00	2009.78
100.00%	239194.00	2810.72

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

La línea de tendencia se comporta como la complejidad obtenida, esto se debe a que hay dos for anidados en el código. Para desarrollar este requerimiento no había una solución exacta en disclib así que optamos por utilizar diferentes estructuras para solucionarlo.

Requerimiento 7

Descripción

```
def req_7(data_structs, fecha1, fecha2, temp_max, temp_min):
    """
    Función que soluciona el requerimiento 7
    """
    # TODO: Realizar el requerimiento 7
    nuevo_grafo = gr.newGraph(datastructure='ADJ_LIST',
                                            directed=True,
                                            size=14000)
    todos_lobos = lt.newList()
    longi_latis = lt.newList()
    lista_intervalo = om.values(data_structs["Fechas"], fecha1, fecha2)
    #z=1
    for cada_fecha in lt.iterator(lista_intervalo):
        #om_temps = cada_fecha["Om_temperaturas"]
        #print(om_temps)
        #for cada_temp in lt.iterator(om.values(om_temps, temp_max, temp_min)): #ACA ESTA RARO
        #if float(lt.getElement(cada_fecha["Datos_fecha"],z)["external-temperature"]) <=temp_max or float(lt.getElem
        for evento in lt.iterator(cada_fecha["Datos_fecha"]):
            if float(evento["external-temperature"]) <=temp_max or float(evento["external-temperature"])>=temp_min:
                lt.addLast(todos_lobos, evento)
                lon_lat_lobo = formato_coo_id(evento)
                lon_lat = formato_coo(evento)
                lt.addLast(longi_latis, lon_lat)
                add_vertice(nuevo_grafo, lon_lat_lobo) #AÑADIR VERTICES DE SEGUIMIENTO
        #z+=1
    print(lt.size(todos_lobos))
    a = 1
    b = 2
    while b <= lt.size(todos_lobos): #AÑADIR ARCOS ENTRE P.SEGUIMIENTO
        if lt.size(todos_lobos) > 1:
            vertice_A = (lt.getElement(todos_lobos, a))
            vertice_B = (lt.getElement(todos_lobos, b))
            distancia = distancia_fn(vertice_A, vertice_B)
            if distancia > 0:
                vertice_A = formato_coo_id(vertice_A)
                vertice_B = formato_coo_id(vertice_B)
                edge = gr.getEdge(nuevo_grafo, vertice_A, vertice_B)
                if edge is None:
                    gr.addEdge(nuevo_grafo, vertice_A, vertice_B, distancia)
        a+=1
        b+=1
    c = 1
    mp_puntos = mp.newMap(20,maptype="PROBING",loadfactor=0.5)
    for lon_lat in lt.iterator(longi_latis):
        data = lt.getElement(todos_lobos, c)
        existe_punto = mp.contains(mp_puntos, lon_lat)
        if existe_punto:
            entry = mp.get(mp_puntos, lon_lat)
            lobo_id = me.getValue(entry)
        else:
            lobo_id = lt.newList("ARRAY_LIST")
            mp.put(mp_puntos, lon_lat, lobo_id)
        existe = False
```

```
        existe = False
        for a in lt.iterator(lobo_id):
            coordenadas = formato_coo(a)
            if lon_lat == coordenadas and data["individual-local-identifier"] == a["individual-local-identifier"] and data[
                existe = True
        if not existe:
            lt.addLast(lobo_id, data)
        if c > lt.size(todos_lobos):
            break
        c+=1
    puntos_encuentr = lt.newList("ARRAY_LIST")
    llaves_p = mp.keySet(mp_puntos) #AÑADIR PUNTOS DE ENCUENTRO
    for llave_p in lt.iterator(llaves_p):
        coor = mp.get(mp_puntos, llave_p)
        lista_coor = me.getValue(coor)
        if lt.size(lista_coor) > 1:
            lt.addLast(puntos_encuentr, llave_p)
    for cada_lon_lat in lt.iterator(puntos_encuentr):
        add_vertice(nuevo_grafo, cada_lon_lat)
    llaves_p = mp.keySet(mp_puntos) #CONECTAR PUNTOS DE ENCUENTRO
    for llave_p in lt.iterator(llaves_p):
        coor = mp.get(mp_puntos, llave_p)
        lista_coor = me.getValue(coor)
        if lt.size(lista_coor) > 1:
            for dato in lt.iterator(lista_coor):
                comparar = formato_coo(dato)
                verticeA = formato_coo_id(dato)
                if comparar == llave_p:
                    edge = gr.getEdge(nuevo_grafo, llave_p, verticeA)
                    if edge is None:
                        gr.addEdge(nuevo_grafo, llave_p, verticeA, 0)
                    edge2 = gr.getEdge(nuevo_grafo, verticeA, llave_p)
                    if edge2 is None:
                        gr.addEdge(nuevo_grafo, verticeA, llave_p, 0)

    #print(gr.numVertices(nuevo_grafo))
    #print(gr.numEdges(nuevo_grafo))
    conectados, tabular = req_8_3(nuevo_grafo, data_structs)
    return gr.numVertices(nuevo_grafo), gr.numEdges(nuevo_grafo), conectados, tabular
```

Entrada	- El data_structs - Fecha inicial - Fecha final - Temperatura máxima - Temperatura mínima
Salidas	El total de manadas reconocidas por sus movimientos y puntos de encuentro (componentes conectados) en el rango de fechas y temperatura ambiente dados. También, retorna los tres primeros y tres últimas manadas con mayor dominio sobre el territorio (de mayor a menor número de puntos de encuentro dentro del componente conectado) con la siguiente información:
Implementado (Sí/No)	Sí

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Para un k<n siendo n el número total de datos. Esto es porque se tomó una cantidad de datos específicos, solo se accedió a los años y mes pasados por parámetro.

Pasos	**Complejidad**
Creacion Array_list	O(1)
New_graph	O(1)
Mp.get	O(1)
Addedge	O(1)
For	O(k)
For anidado	O(k^2)
AddLast	O(1)
Comparacion	O(k)
Kosaraju	O(V*E)
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).

Procesadores	**11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz 2.42 GHz**
Memoria RAM	8 GB
Sistema Operativo	Windows 10

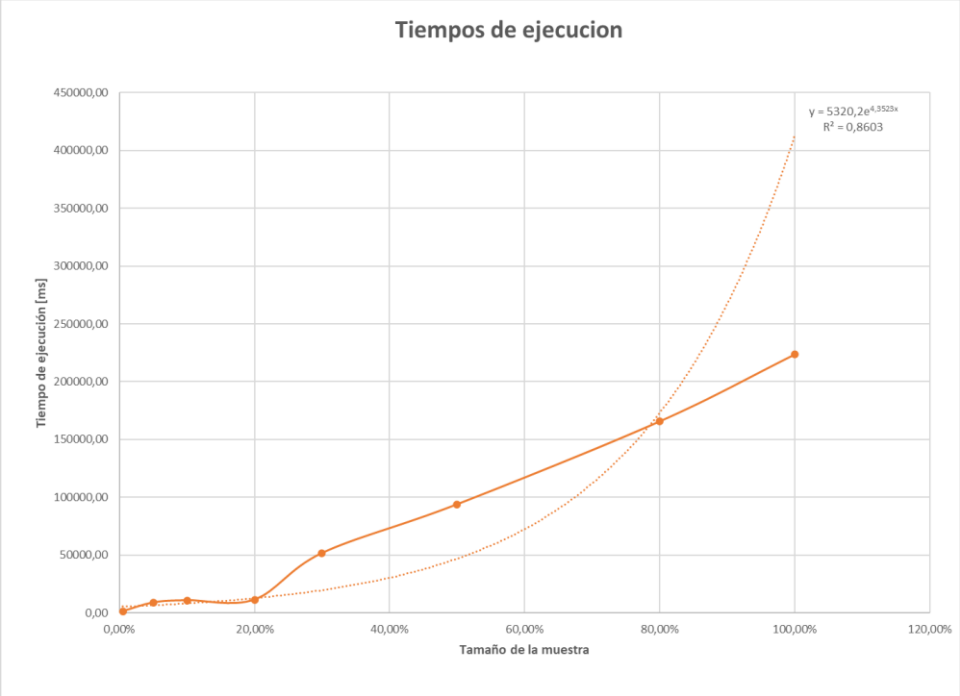
Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Porcentaje de la muestra [pct]	Tamaño de la muestra	Tiempo [ms]
0,50%	2391,00	1268,44
5,00%	11959,00	8846,85
10,00%	23919,00	10813,68
20,00%	47838,00	11353,67
30,00%	71758,00	51628,22
50,00%	119597,00	93850,88
80,00%	191355,00	165617,72
100,00%	239194,00	223518,80

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

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

Es evidente que la gráfica es similar a la del requerimiento 3 debido a que se utiliza el algoritmo kosaraju para encontrar los elementos fuertemente conectados, para así deducir las manadas con más y menos dominio en el territorio. La complejidad es casi lineal, por lo que una complejidad de O(E*V) esta parcialmente correcta