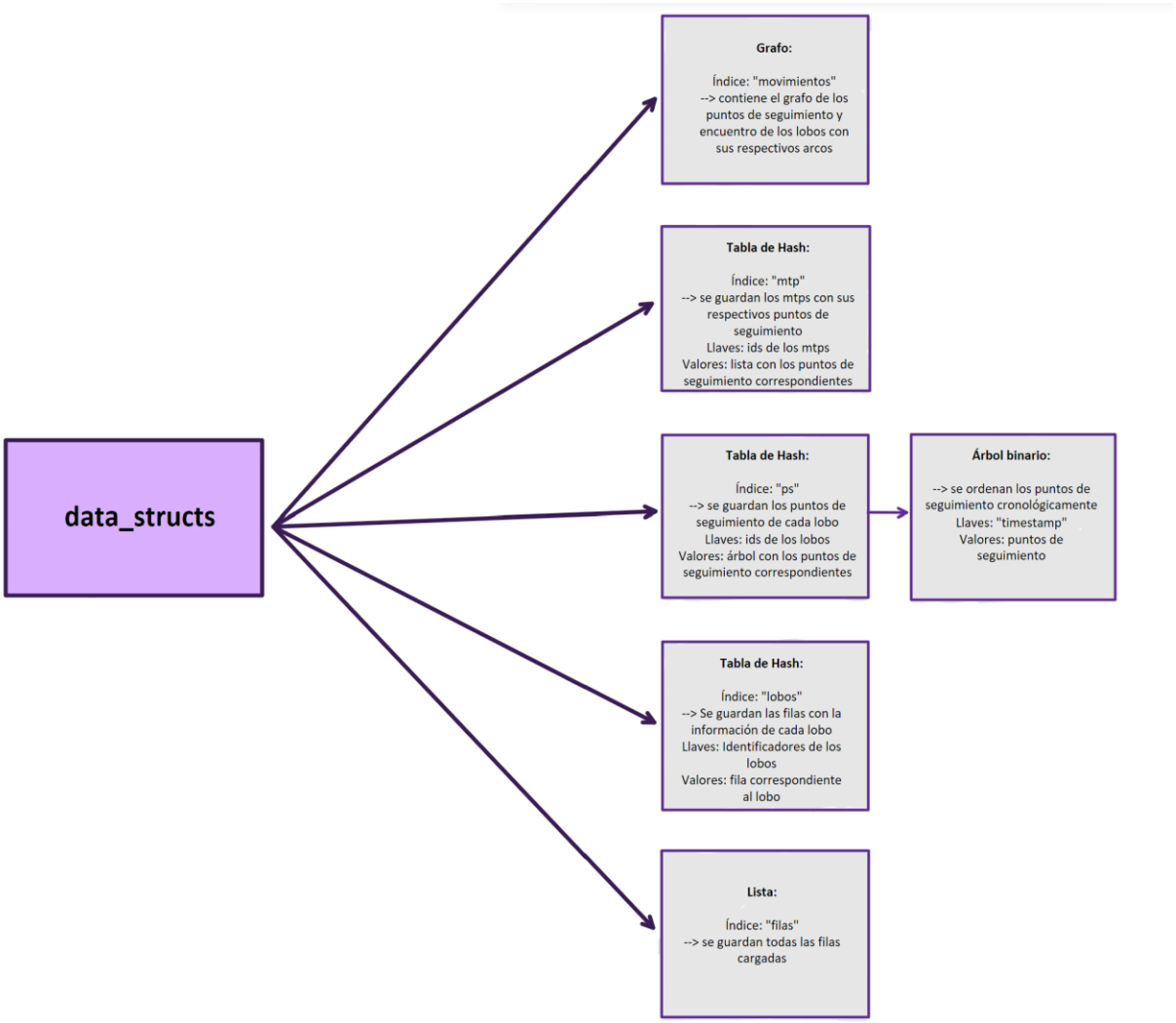
ANÁLISIS DEL RETO

Laura Rodríguez, 202221423, l.rodriguezs2@uniandes.edu.co
Ariadna Vargas, 202221669, at.vargasc1@uniandes.edu.co

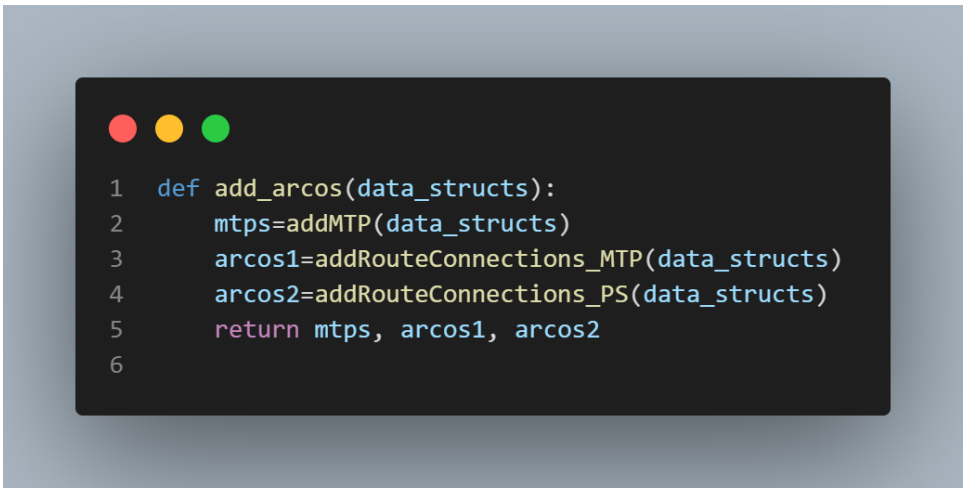
Diagrama carga de datos



Carga de datos

```python
def new_data_structs():
    """
    Inicializa las estructuras de datos del modelo. Las crea de
    manera vacía para posteriormente almacenar la información.
    """
    data_structs={"movimientos":None, "mtp":None, "ps":None, "lobos":None, "filas": None}

    data_structs["movimientos"]=gr.newGraph(datastructure='ADJ_LIST', directed=True, size=180000, cmpfunction=compareID)

    data_structs["mtp"]=mp.newMap(numelements=260001,maptype='PROBING')

    data_structs["ps"]=mp.newMap(numelements=97,maptype='PROBING')

    data_structs["lobos"]= mp.newMap(numelements= 97, maptype="PROBING")

    data_structs["filas"]=lt.newList(datastructure="ARRAY_LIST")

    return data_structs

```

```python
def add_track(data_structs, fila):
    """
    Función para agregar nuevos elementos a la lista
    """
    fila["location-long"]=str(round(float(fila["location-long"]),3))
    fila["location-lat"]=str(round(float(fila["location-lat"]),3))
    add_by_PS(data_structs, fila)
    addMTP_connection(data_structs, fila)
    ps=addPuntoSeguimiento(data_structs, fila)
    lt.addLast(data_structs["filas"], fila)

    return ps

```



```python
def add_arcos(data_structs):
    mtps=addMTP(data_structs)
    arcos1=addRouteConnections_MTP(data_structs)
    arcos2=addRouteConnections_PS(data_structs)
    return mtps, arcos1, arcos2

```

```python
def addMTP(data_structs):
    llaves=mp.keySet(data_structs["mtp"])
    mtps=lt.newList(datastructure= "ARRAY_LIST")
    for llave in lt.iterator(llaves):
        entry=mp.get(data_structs["mtp"], llave)
        value=me.getValue(entry)

        if lt.size(value)>=2:
            lista_lobos= lobos_diferentes(value)
            if lista_lobos== True:
                id=me.getKey(entry)
                lt.addLast(mtps, {"id": id, "info":value})
                if not gr.containsVertex(data_structs["movimientos"], id):
                    gr.insertVertex(data_structs["movimientos"], id)
            else:
                mp.remove(data_structs["mtp"], llave)
        else:
             mp.remove(data_structs["mtp"], llave)
    return mtps

def lobos_diferentes(lista):

    lista_lobos= lt.newList(datastructure= "ARRAY_LIST")

    for lobo in lt.iterator(lista):
        id= lobo["fila"]["individual-local-identifier"]
        if lt.isPresent(lista_lobos, id) == 0:
            lt.addLast(lista_lobos, id)
            if lt.size(lista_lobos)>=2:
                return True
    return False

```

```python
def addPuntoSeguimiento(data_structs,fila):
    id=crear_PuntoSeguimiento(fila)
    if not gr.containsVertex(data_structs["movimientos"], id):
        gr.insertVertex(data_structs["movimientos"], id)
        ps= {"id": id, "info":fila}

        return ps

    return None

```

```python
def addMTP_connection(data_structs, fila):

    llave_MTP= crear_MTP(fila)
    llave_PS= crear_PuntoSeguimiento(fila)
    info={"ps":llave_PS, "fila":fila}
    entry = mp.get(data_structs["mtp"], llave_MTP)
    if entry is None:
        lstroutes = lt.newList(datastructure="ARRAY_LIST", cmpfunction=compareID5)
        lt.addLast(lstroutes, info)
        mp.put(data_structs["mtp"], llave_MTP, lstroutes)
    else:
        lstroutes = entry['value']
        if lt.isPresent(lstroutes, info)==0:
            lt.addLast(lstroutes, info)



def add_by_PS(data_structs, fila):

    llave_ID= fila["individual-local-identifier"]+"_"+ fila["tag-local-identifier"]
    llave_PS= crear_PuntoSeguimiento(fila)
    info={"ps":llave_PS, "fila":fila}

    entry = mp.get(data_structs["ps"], llave_ID)

    if entry is None:
        newentry = om.newMap(omaptype="RBT", cmpfunction=cmp_by_hora)
        mp.put(data_structs["ps"], llave_ID, newentry)
    else:
        newentry = me.getValue(entry)

    llave2=fila["timestamp"]

    om.put(newentry, llave2, info)



def addConnection(data_structs, inicio, final, distancia):

    edge = gr.getEdge(data_structs["movimientos"], inicio, final)
    if edge is None:
        gr.addEdge(data_structs["movimientos"], inicio, final, distancia)


def addRouteConnections_MTP(data_structs):
    contador=0
    lststops = mp.keySet(data_structs["mtp"])
    for key in lt.iterator(lststops):
        lstroutes = mp.get(data_structs["mtp"],key)['value']
        for route in lt.iterator(lstroutes):
            if gr.containsVertex(data_structs["movimientos"], route["ps"]):
                    addConnection(data_structs, key , route["ps"], 0)
                    addConnection(data_structs, route["ps"] , key, 0)
                    contador+=2
    return contador



def addRouteConnections_PS(data_structs):
    contador=0

    lststops = mp.keySet(data_structs["ps"])
    for key in lt.iterator(lststops):
        arbol = mp.get(data_structs["ps"], key)['value']
        lstroutes=om.keySet(arbol)
        prevrout_fila = None
        prevrout_llave= None
        for route in lt.iterator(lstroutes):
            entry=om.get(arbol, route)
            valor=me.getValue(entry)

            route_llave=valor["ps"]
            route_fila= valor["fila"]
            if prevrout_llave is not None:
                distancia= calcular_distancia(prevrout_fila, route_fila)
                if distancia != 0:

                    addConnection(data_structs, prevrout_llave, route_llave, distancia)
                    contador+=1
            prevrout_llave = route_llave
            prevrout_fila= route_fila

    return contador


def calcular_distancia(fila1, fila2):

    lat_1= float(fila1["location-lat"])
    lon_1= float(fila1["location-long"])

    lat_2= float(fila2["location-lat"])
    lon_2= float(fila2["location-long"])

    lat_1=lat_1*np.pi/180
    lon_1=lon_1*np.pi/180
    lat_2=lat_2*np.pi/180
    lon_2=lon_2*np.pi/180

    a=(np.sin((lat_2- lat_1)/2))**2

    b=(np.sin((lon_2-lon_1)/2))**2

    c=a+np.cos(lat_2)*np.cos(lat_1)*b

    d= 2* np.arcsin(np.sqrt(c))*6371

    return round(d,3)

```

```
def addLobo (data_structs, fila):
    llave_lobo= fila["animal-id"]+"_"+fila["tag-id"]
    mp.put(data_structs["lobos"], llave_lobo, fila)

```

Descripción

Entrada	data_structs, nombre_archivo
Salidas	Cantidad lobos, tamaño 1er archivo, cantidad mtp, cantidad arcos para mtp, cantidad de ps, cantidad de arcos para ps, ps, tiempo
Implementado (Sí/No)	Si, grupal

Parte 1: Análisis de complejidad (carga de datos archivo tracks)

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1: Por cada fila se agrega el punto de seguimiento en el mapa data_structs[“ps”] en el lobo correspondiente. El punto de seguimiento se agrega dentro de un árbol ordenado cronológicamente	O(N log k) donde k es la cantidad de ps en cada lobo y N es la cantidad de filas
Paso 2: Por cada fila se agrega en el mapa dta_structs[“mtp”] se agrega el punto de seguimiento al posible mtp correspondiente. El ps se agrega en una lista.	O(N)
Paso 3: Por cada fila se agrega el ps al grafo	O(N)
Paso 4: Por cada fila se agrega la fila a la lista data_structs[“filas”]	O(N)
Paso 5: A partir del mapa data_structs[“mtp”] se recorren los ps de cada mtp para verificar que haya dos lobos en el mtp.	O(m*n) donde m es la cantidad de filas cada mtp y n es la cantidad de mtp. n<m<N
Paso 6: Por cada mtp de data_structs[“mtp”] se agrega las conexiones entre el mtp y sus ps correspondientes.	O(m*n)
Paso 7: A partir de data_structs[“ps”] se recorren los árboles de cada lobo para crear un arco entre los ps consecutivos (verificando que el lobo se haya movido)	O(log k+k)
TOTAL	***O(N log k)***

Parte 2: Análisis de complejidad (carga de datos archivo individuals)

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1: Por cada fila se agrega al mapa data_structs[“lobos”] la fila correspondiente al lobo	O(1), el máximo son 46 lobos entonces es una constante en todos los archivos
TOTAL	***O(1)***

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).

	Máquina 1	Máquina 2
Procesadores	12th Gen Intel(R) Core(TM) i7-1255U 1.70 GHz	AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx 2.10 GHz
Memoria RAM (GB)	16,0 GB (15,7 GB utilizable)	16.0 GB (13.9 GB usable)
Sistema Operativo	Sistema operativo de 64 bits, procesador x64, Windows 11 Home Single Language	Windows 11

Máquina 1:

Entrada	**Datos entrada**	**Tiempo (ms)**
small	66	1124,966
5 pct	468	3948,908
10 pct	934	9367,182
20 pct	1821	18210,663
30 pct	2705	38280,925
50 pct	4636	123782,728
80 pct	7645	151480,256
large	9642	218956,756

Máquina 2:

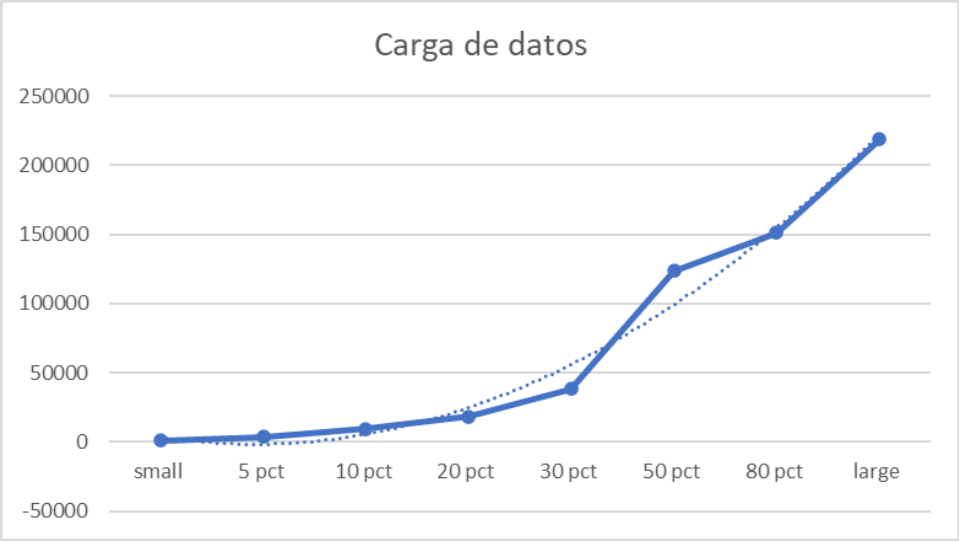
Entrada	**Datos entrada**	**Tiempo (ms)**

small	66	3125.336
5 pct	468	11880.791
10 pct	934	27145.158
20 pct	1821	49073.155
30 pct	2705	79035.982
50 pct	4636	182928.2
80 pct	7645	260177.631
large	9642	293926.59

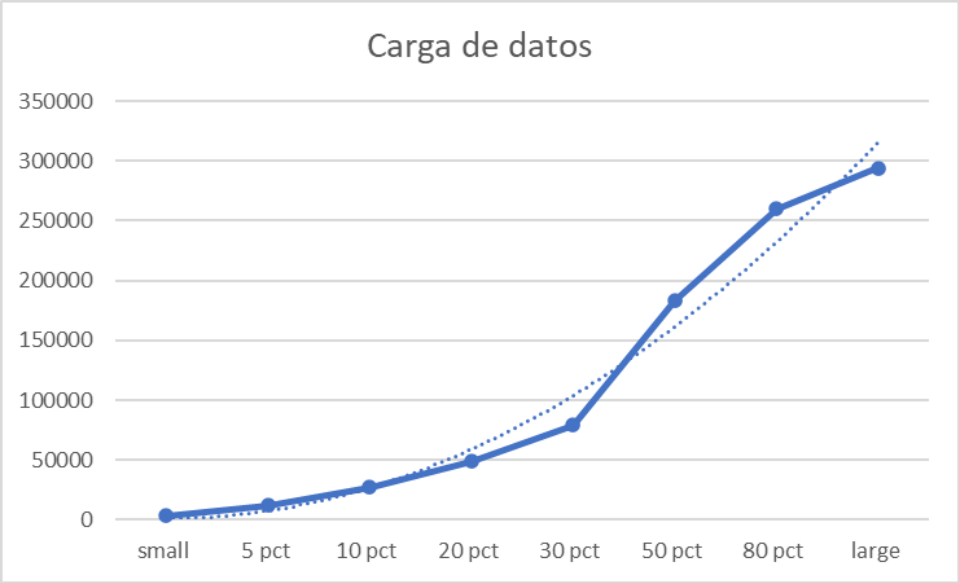
Gráficas

Las gráficas con la representación de las pruebas realizadas.

Máquina 1:



Máquina 2:



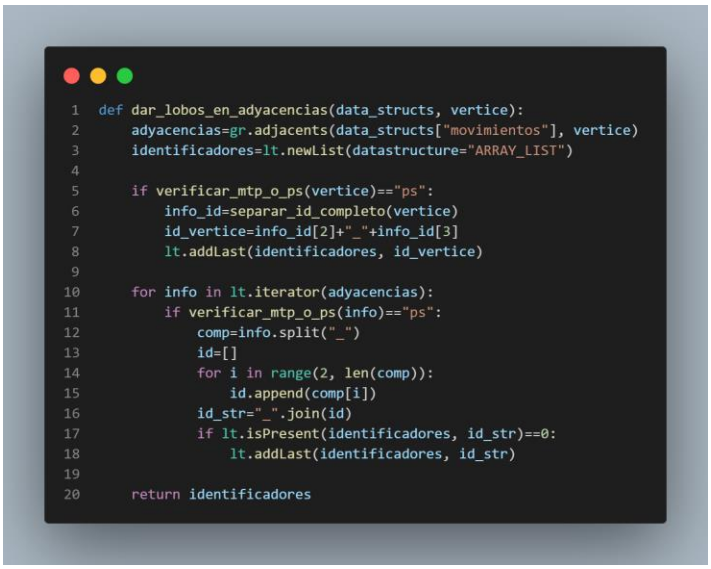
Análisis

La carga de datos tiene una complejidad de O(N log K) porque se agrega los puntos de seguimiento a un árbol RBT, lo cual tiene una complejidad de O(log k) donde k son la cantidad de filas correspondientes a cada lobo. Como se agrega un punto de seguimiento por cada fila, la complejidad total va a ser N* O(log k) = O(N log k). Este comportamiento se puede evidenciar en las gráficas de tiempo, puesto que estás muestran un comportamiento similar al logarítmico.

Requerimiento <<1>>

```python
def req_1(data_structs, origen, destino):
    """
    Función que soluciona el requerimiento 1
    """
    if gr.containsVertex(data_structs["movimientos"], origen) and gr.containsVertex(data_structs["movimientos"], destino):
        search=dfs.DepthFirstSearch(data_structs["movimientos"], origen)

        a=dfs.hasPathTo(search, destino)

        if a is True:
            camino=dfs.pathTo(search, destino)

            mtps=0
            pss=0
            distancia=0
            for i in range(1, lt.size(camino)+1):
                vertice=lt.getElement(camino, i)
                if verificar_mtp_o_ps(vertice)=="mtp":
                    mtps+=1
                else:
                    pss+=1

                if i<=lt.size(camino)-1:
                    vertice2=lt.getElement(camino, i+1)

                    arco=gr.getEdge(data_structs["movimientos"], vertice2, vertice)
                    distancia+=e.weight(arco)

            info_filas=lt.newList(datastructure="SINGLE_LINKED")

            for i in range(1, 6):
                lista=lt.newList(datastructure="ARRAY_LIST")
                llave=lt.getElement(camino, i)
                lt.addLast(lista, llave)
                f=devolver_formato(llave)
                ubicacion=separar_id(f)
                lt.addLast(lista,ubicacion)
                ids=dar_lobos_en_adyacencias(data_structs, llave)
                lt.addLast(lista,lt.size(ids))
                adyacencias=gr.adjacents(data_structs["movimientos"], llave)

                lt.addFirst(info_filas, lista)

            for i in range(lt.size(camino)-5, lt.size(camino)+1):
                lista=lt.newList(datastructure="ARRAY_LIST")
                llave=lt.getElement(camino, i)
                lt.addLast(lista, llave)
                f=devolver_formato(llave)
                ubicacion=separar_id(f)
                lt.addLast(lista,ubicacion)
                ids=dar_lobos_en_adyacencias(data_structs, llave)
                lt.addLast(lista,lt.size(ids))
                adyacencias=gr.adjacents(data_structs["movimientos"], llave)

                lt.addFirst(info_filas, lista)


            mapa=crear_mapa_folium()

            trail=[]


            for nodo in lt.iterator(camino):
                crear_marcador(mapa, nodo)
                crear_ruta(trail, nodo)

            agregar_ruta(mapa, trail)

            mapa.show_in_browser()



            return distancia, mtps, pss, info_filas
        else:
            return None

```

```python
def verificar_mtp_o_ps(vertice):
    if vertice.count("_")==1:
        return "mtp"

    else:
        return "ps"
```



```python
def dar_lobos_en_adyacencias(data_structs, vertice):
    adyacencias=gr.adjacents(data_structs["movimientos"], vertice)
    identificadores=lt.newList(datastructure="ARRAY_LIST")

    if verificar_mtp_o_ps(vertice)=="ps":
        info_id=separar_id_completo(vertice)
        id_vertice=info_id[2]+"_"+info_id[3]
        lt.addLast(identificadores, id_vertice)

    for info in lt.iterator(adyacencias):
        if verificar_mtp_o_ps(info)=="ps":
            comp=info.split("_")
            id=[]
            for i in range(2, len(comp)):
                id.append(comp[i])
            id_str="_".join(id)
            if lt.isPresent(identificadores, id_str)==0:
                lt.addLast(identificadores, id_str)

    return identificadores
```

```python
def separar_id_completo(dato):

    componentes=[]
    inicio=0
    a=dato.count("_")
    if a==4:
        contador=0
        for i in range(0, len(dato)):
            if contador==2:
                if dato[i] == "_":
                        contador+=1
            else:
                if len(componentes)==3:
                    info=dato[inicio:]
                    componentes.append(info)
                else:
                    if dato[i] == "_":
                        contador+=1
                        info=dato[inicio:i]
                        inicio=i+1
                        componentes.append(info)


    else:
        for i in range(0, len(dato)):
            if len(componentes)==3:
                info=dato[inicio:]
                componentes.append(info)
            else:
                if dato[i] == "_":
                    info=dato[inicio:i]
                    inicio=i+1
                    componentes.append(info)

    return componentes
```

Descripción

Este requerimiento se encarga de conocer si existe un camino utilizado entre dos puntos de encuentro para lobos. Obteniendo la distancia total entre el camino, los puntos de encuentro y los puntos de seguimiento dentro del camino. Y los primeros y últimos 5 nodos en el camino.

Entrada	Data_structs, origen, destino
Salidas	Distancia(distancia total entre los dos puntos) , mtp (total puntos de encuentro), ps (total puntos de seguimiento), info_filas (lista con las filas correspondientes de los vertices)
Implementado (Sí/No)	Si, grupal

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1: Verifica si el mtp de origen y destino están dentro del grafo	O(1)
Paso 2: Se hace DFS con el mtp de origen	O(V+A)

Paso 3: Se pregunta si existe un camino entre el origen y el destino	O(1)
Paso 4: Se hace el camino entre el origen y el destino	O(V)
Paso 5: se recorre cada vértice en el camino y se verifica si es un mtp o un ps. Además, va guardando la distancia entre un vértice y el otro (peso).	O(k*c), c<A, donde c es la cantidad de adyacentes que tiene el primer vértice. Donde k es la cantidad de vértices en el camino encontrado. K<A
Paso 6: Se sacan los 5 primeros y 5 últimos vértices dentro del camino.	O(c), c<A, donde c es la cantidad de adyacentes que tiene el vértice
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).

	Máquina 1	Máquina 2
Procesadores	12th Gen Intel(R) Core(TM) i7-1255U 1.70 GHz	AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx 2.10 GHz
Memoria RAM (GB)	16,0 GB (15,7 GB utilizable)	16.0 GB (13.9 GB usable)
Sistema Operativo	Sistema operativo de 64 bits, procesador x64, Windows 11 Home Single Language	Windows 11

Máquina 1:

Entrada	Datos entrada	Tiempo (ms)
small	66	10,827
5 pct	468	125,153
10 pct	934	167,964
20 pct	1821	273,919
30 pct	2705	2260,04
50 pct	4636	3258,46
80 pct	7645	5663,5
large	9642	6726,77

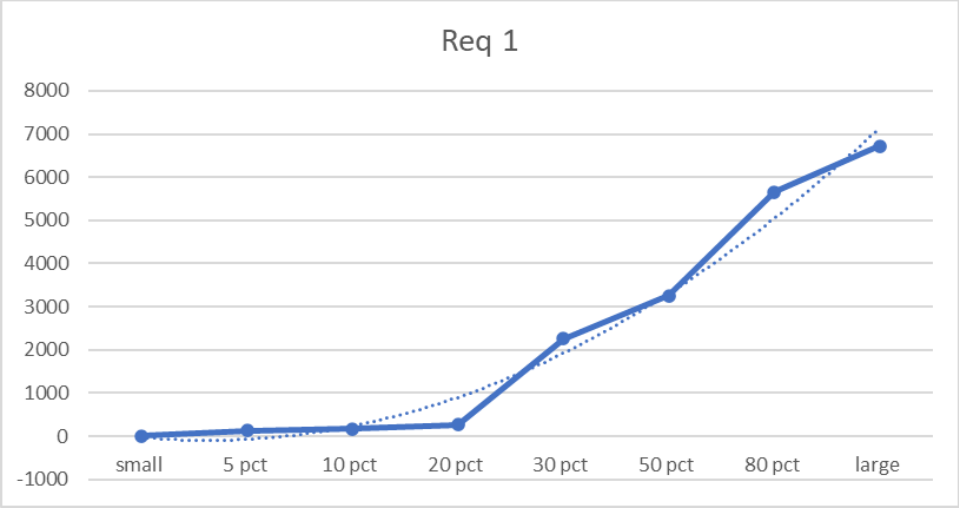
Máquina 2:

Entrada	Datos entrada	Tiempo (ms)
small	66	26.728
5 pct	468	215.43
10 pct	934	326.427
20 pct	1821	504.17
30 pct	2705	4349.66
50 pct	4636	6409.02
80 pct	7645	10869.1
large	9642	15799.5

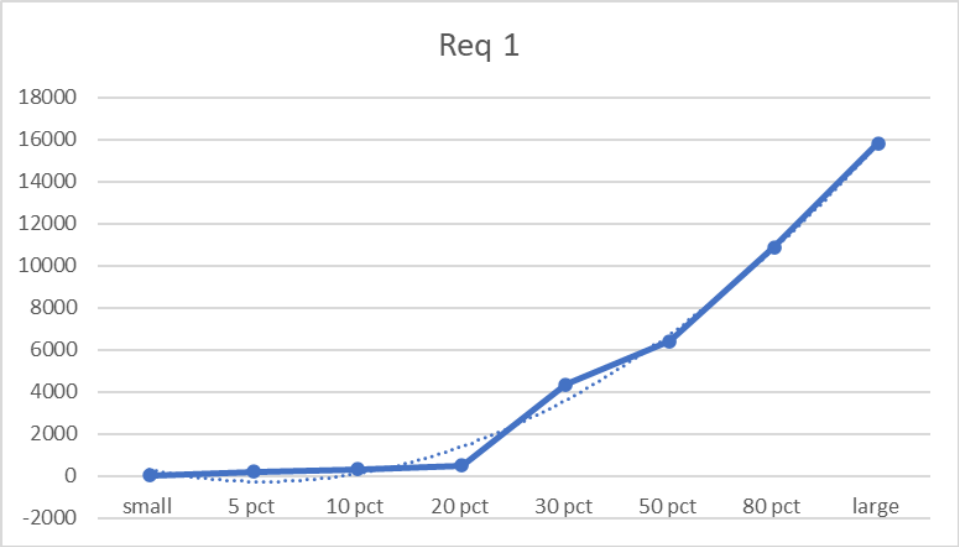
Gráficas

Las gráficas con la representación de las pruebas realizadas.

Máquina 1:



Máquina 2:



Análisis

Este requerimiento tiene una complejidad de O(V + A) porque se hace un dfs con el grafo, el cual tiene esta complejidad. Teniendo en cuenta que tanto V como A son variables y puede que no crezcan al mismo ritmo, las gráficas muestran un comportamiento adecuado ya que se evidencia un comportamiento lineal o un poco mayor.

Requerimiento <<3>>

```python
def req_3(data_structs):
    """
    Función que soluciona el requerimiento 3
    """
    # TODO: Realizar el requerimiento 3

    search = scc.KosarajuSCC(data_structs["movimientos"])

    cantidad = scc.connectedComponents(search)

    keyset_ref = mp.keySet(data_structs["mtp"])
    llave_ref= lt.getElement(keyset_ref, 1)
    cfc= scc.sccCount(data_structs["movimientos"], search, llave_ref)

    mapa_cfc = mp.newMap(numelements= cantidad//4 , maptype= "CHAINING", loadfactor= 4 )
    vertices = mp.keySet(cfc["idscc"])


    #reversar
    for vertice in lt.iterator(vertices):
        value = mp.get(cfc["idscc"], vertice)
        id=value["value"]
        entry = mp.get(mapa_cfc, id)
        if entry == None:
            lista = lt.newList(datastructure="ARRAY_LIST")
            mp.put(mapa_cfc, id, lista)
        else:
            lista= mp.get(mapa_cfc, id )["value"]
        lt.addLast(lista, vertice)


    #encontrar 5 mayores
    keyset_cfc= mp.keySet(mapa_cfc)
    mayor_cfc= mp.newMap(numelements= 11, maptype="PROBING" )
    lista_mayores = lt.newList("ARRAY_LIST")

    while mp.size(mayor_cfc) < 5:
        mayor=0
        for llave in lt.iterator(keyset_cfc):
            value= mp.get(mapa_cfc, llave)["value"]
            tamanio= lt.size(value)
            if tamanio> mayor and mp.contains(mayor_cfc, llave) == False:
                mayor= tamanio
                mayor_values = value
                llave_mayor = llave
        mp.put(mayor_cfc, llave_mayor, mayor_values)
        lt.addLast(lista_mayores, llave_mayor)


    keyset_mayor_cfc = mp.keySet(mayor_cfc)
    mapa = mp.newMap(numelements= 11, maptype="PROBING", loadfactor=0.5)


    for llave in lt.iterator(keyset_mayor_cfc):

        diccionario = {}
        mp.put(mapa, llave, diccionario)

```

```python
#encontrar 3 primeros y 3 ultimos

for llave in lt.iterator(keyset_mayor_cfc):
    primeros_ultimos = lt.newList("ARRAY_LIST")
    value= mp.get(mayor_cfc, llave)["value"]
    if lt.size(value)<=6:
        lt.addLast(primeros_ultimos, value)

    else:
        for i in range (1,4):
            fila= lt.getElement(value, i)
            lt.addLast(primeros_ultimos,  fila)
        for i in range (lt.size(value)-2, lt.size(value)+1):
            fila= lt.getElement(value, i)
            lt.addLast(primeros_ultimos,  fila)


    dicc=mp.get(mapa, llave)["value"]
    dicc["primeros_ultimos"]= primeros_ultimos

#tamaño

for llave in lt.iterator(keyset_mayor_cfc):
    dicc = mp.get(mapa, llave)["value"]
    value= mp.get(mayor_cfc, llave)["value"]
    dicc["tamaño"]= lt.size(value)

#encontrar lat y long

for llave in lt.iterator(keyset_mayor_cfc):

    value= mp.get(mayor_cfc, llave)["value"]
    mayor = buscar_mayor_long_lat_lista(value)
    menor = buscar_menor_long_lat_lista (value)
    lat_max  = lt.getElement(mayor, 2)
    long_max = lt.getElement(mayor, 1)
    lat_min = lt.getElement(menor, 2)
    long_min = lt.getElement(menor, 1)

    dicc = mp.get(mapa, llave)["value"]
    dicc["min_lat"] = lat_min
    dicc["max_lat"] = lat_max
    dicc["min_lon"] = long_min
    dicc["max_lon"] = long_max

#encontrar lobos

for llave in lt.iterator(keyset_mayor_cfc):

    lista_lobos = lt.newList(datastructure= "ARRAY_LIST")
    value= mp.get(mayor_cfc, llave)["value"]
    lobos = dar_lobos_en_adyacencias_lista(value)
    cantidad_lobos = lt.size(lobos)


    if cantidad_lobos <=6:
        for lobo in lt.iterator(lobos):
            fila= mp.get(data_structs["lobos"], lobo)["value"]
            lt.addLast(lista_lobos,fila )

    else:
        for i in range (1,4):
            lobo= lt.getElement(lobos, i)
            fila= mp.get(data_structs["lobos"], lobo)["value"]
            lt.addLast(lista_lobos, fila )

        for i in range (lt.size(lobos)-2, lt.size(lobos)+1):
            lobo= lt.getElement(lobos, i)
            fila= mp.get(data_structs["lobos"], lobo)["value"]
            lt.addLast(lista_lobos, fila )


    dicc=mp.get(mapa, llave)["value"]
    dicc["cantidad_lobos"] = cantidad_lobos
    dicc["informacion_lobos"]= lista_lobos

mapa_folium = crear_mapa_folium()
lista_color = ["red", "purple", "pink", "darkgreen", 'cadetblue']
contador = 0
for llave in lt.iterator(keyset_mayor_cfc):
    puntos = mp.get(mayor_cfc, llave)["value"]
    entry= mp.get(mapa, llave)["value"]

    crear_circulo(mapa_folium, entry["min_lon"], entry["max_lon"], entry["min_lat"], entry["max_lat"], llave, lista_color[contador] )
    contador+= 1
    for punto in lt.iterator(puntos):
        crear_marcador(mapa_folium, punto)

mapa_folium.show_in_browser()

return cantidad, mapa, lista_mayores

```

```python
def buscar_mayor_long_lat_lista(lista):

    mayor_id=lt.getElement(lista,1)
    mayor_id_separado=separar_id(mayor_id)
    long_ref= (mayor_id_separado)[0]
    lat_ref= (mayor_id_separado)[1]
    mayor_long_ref= devolver_formato(long_ref)
    mayor_lat_ref= devolver_formato(lat_ref)

    mayor_long=float(mayor_long_ref)

    mayor_info=lt.newList(datastructure="ARRAY_LIST")


    for id in lt.iterator(lista):
        long_id=separar_id(id)[0]
        long= devolver_formato(long_id)
        if float(long)>mayor_long:
            mayor_long=float(long)
            mayor_id=id


    lt.addLast(mayor_info, mayor_long)

    mayor_lat=float(mayor_lat_ref)

    for id in lt.iterator(lista):
        lat_id=separar_id(id)[1]
        lat = devolver_formato(lat_id)
        if float(lat)>mayor_lat:
            mayor_lat=float(lat)
            mayor_id=id


    lt.addLast(mayor_info, mayor_lat)

    return mayor_info

```

```python
def buscar_menor_long_lat_lista(lista):

    menor_id=lt.getElement(lista,1)
    menor_id_separado=separar_id(menor_id)
    long_ref= (menor_id_separado)[0]
    lat_ref= (menor_id_separado)[1]
    menor_long_ref= devolver_formato(long_ref)
    menor_lat_ref= devolver_formato(lat_ref)

    menor_long=float(menor_long_ref)

    menor_info=lt.newList(datastructure="ARRAY_LIST")

    for id in lt.iterator(lista):
        long_id=separar_id(id)[0]
        long= devolver_formato(long_id)
        if float(long)<menor_long:
            menor_long=float(long)
            menor_id=id


    lt.addLast(menor_info, menor_long)



    menor_lat=float(menor_lat_ref)
    for id in lt.iterator(lista):
        lat_id=separar_id(id)[1]
        lat = devolver_formato(lat_id)
        if float(lat)<menor_lat:
            menor_lat=float(lat)
            menor_id=id


    lt.addLast(menor_info, menor_lat)

    return menor_info

```

Descripción

Este requerimiento se encarga de encontrar las manadas (componentes fuertemente conectados) dentro del habitad.

Entrada	Data_structs
Salidas	Un mapa con la información de las 5 manadas más grandes, la cantidad de manadas que hay, una lista con el orden de mayor a menor de las 5 manadas.
Implementado (Sí/No)	Si, Ariadna Vargas

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1: Se hace Kosaraju para encontrar los cfc	O(V+A)
Paso 2: Se encuentra la cantidad de cfc	O(1)
Paso 3: Se sacan las llaves del mapa de ps y se saca una llave de referencia.	O(V+k)

Paso 4: Con la llave de referencia y los cfc ya encontrados se saca el mapa con los vértices de cada cfc.	O(V)
Paso 5: Se saca el keyset de los vertices de los cfc	O(V+k)
Paso 6: En un mapa se va agregando el id del cfc como llave y su valor como los vértices que pertenecen a dicho cfc.	O(V)
Paso 7: Se encuentran las 5 manadas más grandes (con mayor cantidad de vertices) y se agregan a un mapa	O(1)
Paso 8: Se crea un mapa donde la llave va a ser el id del cfc y su valor un diccionario donde se va a almacenar la información del cfc.	O(1)
Paso 9: Se encuentran los 3 primeros y últimos puntos de encuentro de los 5 cfc más grandes	O(1)
Paso 10: Se encuentra el tamaño de los 5 cfc más grandes	O(1)
Paso 11: Busca la latitud y longitud minina y máxima de los 5 cfc más grandes con funciones auxiliares.	O(v) donde v es la cantidad de vértices que hay en cada componente fuertemente conectado
Paso 12: Encontrar los tres primeros y últimos lobos de los 5 cfc más grandes.	O(v) donde v es la cantidad de vértices que hay en cada componente fuertemente conectado
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).

	Máquina 1	Máquina 2
Procesadores	12th Gen Intel(R) Core(TM) i7-1255U 1.70 GHz	AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx 2.10 GHz
Memoria RAM (GB)	16,0 GB (15,7 GB utilizable)	16.0 GB (13.9 GB usable)
Sistema Operativo	Sistema operativo de 64 bits, procesador x64, Windows 11 Home Single Language	Windows 11

Máquina 1:

Entrada	Datos entrada	Tiempo (ms)
small	66	844,782
5 pct	468	3386,36
10 pct	934	6207,48
20 pct	1821	11589,8
30 pct	2705	14751,4
50 pct	4636	26903,3
80 pct	7645	37707,2
large	9642	43580,8

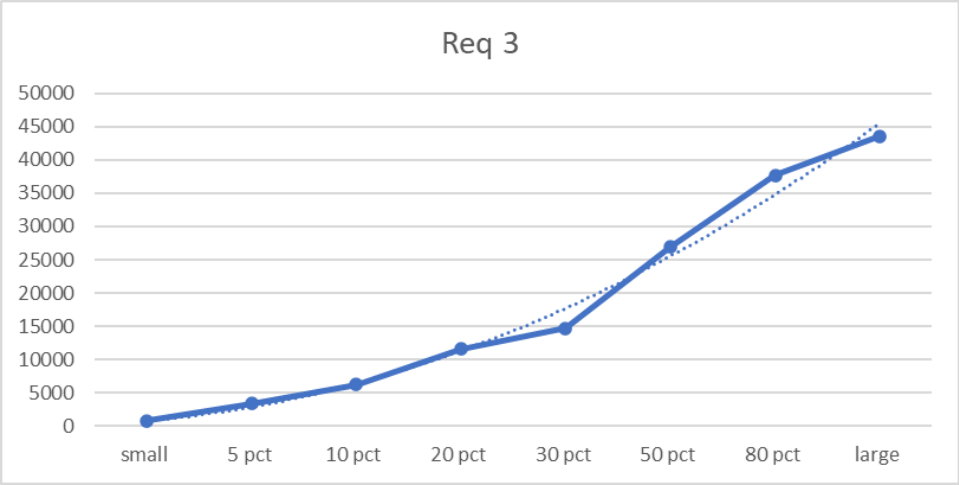
Máquina 2:

Entrada	Datos entrada	Tiempo (ms)
small	66	1351.06
5 pct	468	9772.38
10 pct	934	14955.3
20 pct	1821	22992.7
30 pct	2705	31819.1
50 pct	4636	50116.5
80 pct	7645	61012.8
large	9642	77474.2

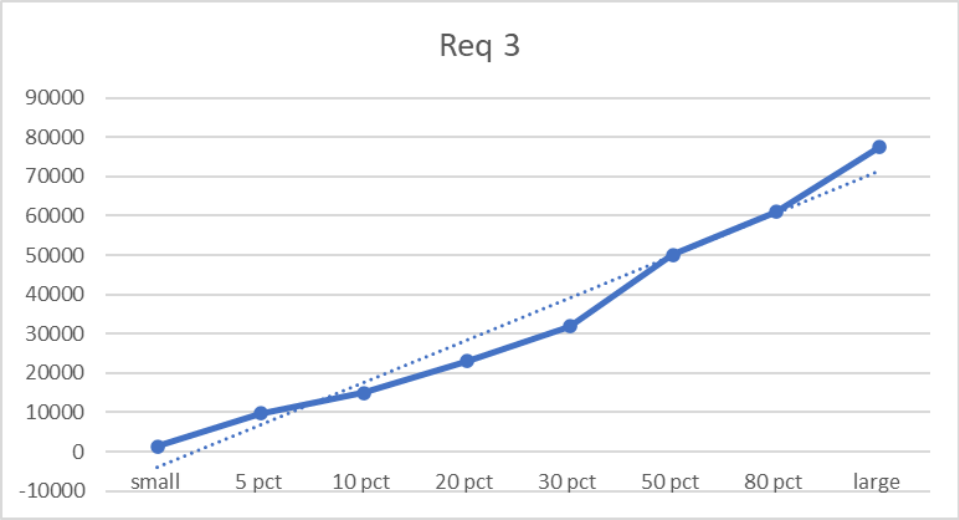
Gráficas

Las gráficas con la representación de las pruebas realizadas.

Máquina 1:



Máquina 2:



Análisis

Este requerimiento tiene una complejidad de O(V +A) porque se realiza un Kosaraju con el grafo para encontrar los componentes fuertemente conectados y Kosaraju tiene una complejidad de O(V +A). Las gráficas muestran un comportamiento lineal o un poco mayor, lo cual coincide con la complejidad calculada.

Requerimiento <<4>>

```python
def req_4(data_structs, origen, destino):
    """
    Función que soluciona el requerimiento 4
    """
    #info_origen
    mtp_origen=encontrar_mtp_mas_cercano(data_structs, origen)
    mtp_llave_origen=mtp_origen[0]
    mtp_dist_origen=mtp_origen[3]

    ubicacion_origen=separar_id(mtp_llave_origen)
    lobos_origen=dar_lobos_en_adyacencias(data_structs, mtp_llave_origen)
    info_origen={"id":mtp_llave_origen, "longitud": devolver_formato(ubicacion_origen[0]), "latitud": devolver_formato(ubicacion_origen[1]), "distancia":mtp_dist_origen, "lobos":lobos_origen}

    #info_destino
    mtp_destino=encontrar_mtp_mas_cercano(data_structs, destino)
    mtp_llave_destino=mtp_destino[0]
    mtp_dist_destino=mtp_destino[3]

    ubicacion_destino=separar_id(mtp_llave_destino)
    lobos_destino=dar_lobos_en_adyacencias(data_structs, mtp_llave_destino)
    info_destino={"id":mtp_llave_destino, "longitud": devolver_formato(ubicacion_destino[0]), "latitud": devolver_formato(ubicacion_destino[1]), "distancia":mtp_dist_destino, "lobos": lobos_destino}

```

```python
search=djk.Dijkstra(data_structs["movimientos"], mtp_llave_origen)


a=djk.hasPathTo(search, mtp_llave_destino)


if a is True:
    distancia=djk.distTo(search, mtp_llave_destino)

    camino=djk.pathTo(search, mtp_llave_destino)

    arcos=lt.newList(datastructure="SINGLE_LINKED")
    for info in lt.iterator(camino):
        lt.addFirst(arcos, info)

    nodos=lt.newList(datastructure="ARRAY_LIST")
    for arco in lt.iterator(arcos):
        if lt.isPresent(nodos, arco["vertexA"])==0:
            lt.addLast(nodos, arco["vertexA"])
        if lt.isPresent(nodos, arco["vertexB"])==0:
            lt.addLast(nodos, arco["vertexB"])

    mtps=lt.newList(datastructure="ARRAY_LIST")
    for nodo in lt.iterator(nodos):
        if verificar_mtp_o_ps(nodo)=="mtp":
            lt.addLast(mtps, nodo)

    lobos_en_camino=dar_lobos_en_adyacencias_lista(nodos)

    mapa=crear_mapa_folium()

    trail=[]
    lat_long_origen=(float(origen["location-lat"]), float(origen["location-long"]))
    trail.append(lat_long_origen)

    for nodo in lt.iterator(nodos):
        crear_marcador(mapa, nodo)
        crear_ruta(trail, nodo)

    lat_long_destino=(float(destino["location-lat"]), float(destino["location-long"]))
    trail.append(lat_long_destino)

    crear_punto(mapa, origen["location-long"], origen["location-lat"])
    crear_punto(mapa, destino["location-long"], destino["location-lat"])
    agregar_ruta(mapa, trail)

    mapa.show_in_browser()

    return info_origen, info_destino, distancia, lobos_en_camino, arcos, nodos, mtps

else:
    return None

```

Descripción

Este requerimiento se encarga de encontrar el camino más corto entre dos puntos en el habitad.

Entrada	Data_structs, origen, destino
Salidas	Información del mtp más cercano al origen, información del mtp más cercano al destino, la distancia recorrida, los lobos que circulan por el camino, los arcos del camino, los nodos del camino, los mtps dentro del camino
Implementado (Sí/No)	Si, Laura Rodríguez

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1: Se busca el mtp más cercano a las coordenadas del origen y del destino	O(k) donde k es la cantidad de mtps que hay en el grafo
Paso 2: Se saca la información del mtp más cercano al origen y al destino para retornarla	O(1)
Paso 3: Se crea un search de Dijkstra con el grafo	O(A log V)
Paso 4: Se verifica que haya un camino entre el mtp de origen y destino	O(1)
Paso 5: Si existe un camino se saca el camino y la distancia del camino	O(V)
Paso 6: Se recorren los arcos del camino para sacar los vértices correspondientes.	O(m) donde m es la cantidad de arcos que tiene el camino (m<A)
Paso 7: Se recorren los vértices para encontrar los vértices que son mtps	O(m)
Paso 8: Se buscan los lobos que circulan el camino	O(m)
TOTAL	***O(A 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).

	Máquina 1	Máquina 2
Procesadores	12th Gen Intel(R) Core(TM) i7-1255U 1.70 GHz	AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx 2.10 GHz
Memoria RAM (GB)	16,0 GB (15,7 GB utilizable)	16.0 GB (13.9 GB usable)
Sistema Operativo	Sistema operativo de 64 bits, procesador x64, Windows 11 Home Single Language	Windows 11

Máquina 1:

Entrada	Datos entrada	Tiempo (ms)
small	66	510,436
5 pct	468	951,169
10 pct	934	6138,44
20 pct	1821	32808,2
30 pct	2705	36753,6
50 pct	4636	47064,7
80 pct	7645	54499,4
large	9642	72932,8

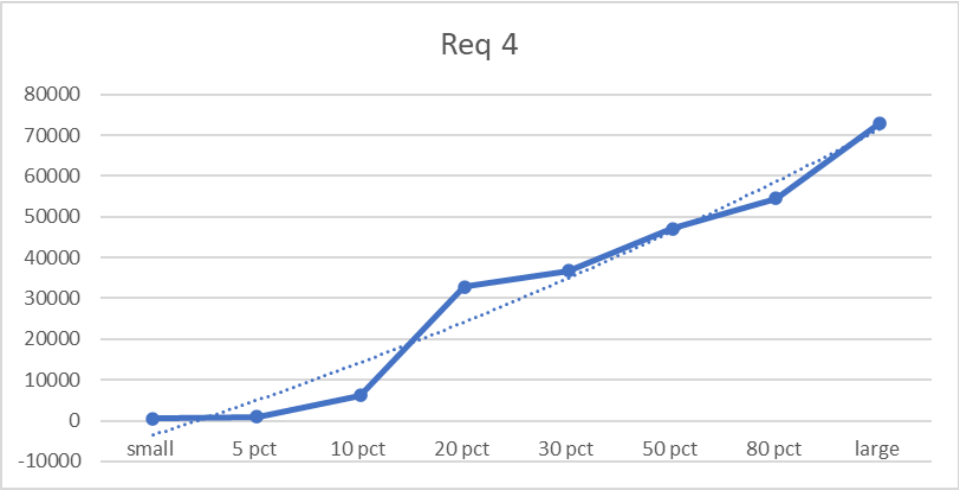
Máquina 2:

Entrada	Datos entrada	Tiempo (ms)
small	66	1048.44
5 pct	468	3279.79
10 pct	934	17847.5
20 pct	1821	43167.8
30 pct	2705	137708
50 pct	4636	153992
80 pct	7645	667485
large	9642	915914

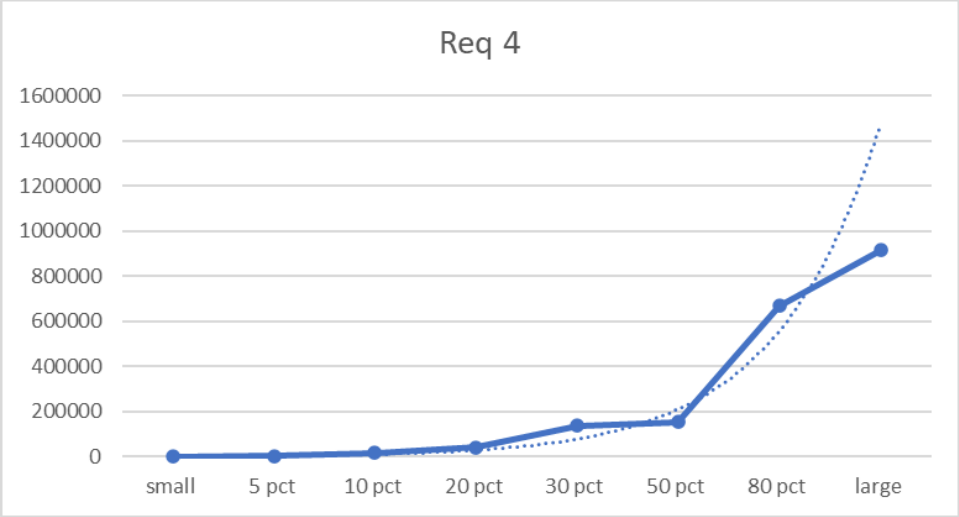
Gráficas

Las gráficas con la representación de las pruebas realizadas.

Máquina 1:



Máquina 2:



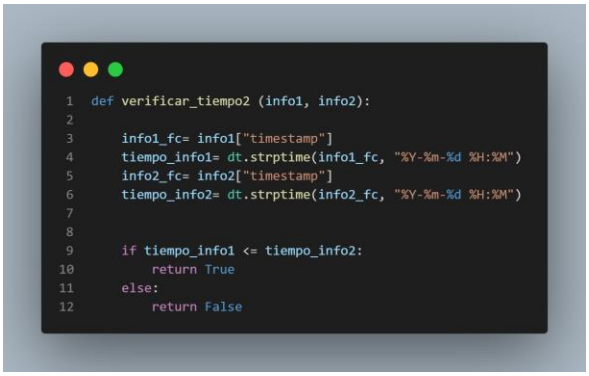
Análisis

Este requerimiento tiene una complejidad de O(A log V) porque se realiza Dijkstra con el grafo de los lobos y Dijkstra tiene esta complejidad. Las gráficas tienen una tendencia que coincide con una gráfica de n log k, puesto que las curvaturas coinciden.

Requerimiento <<6>>

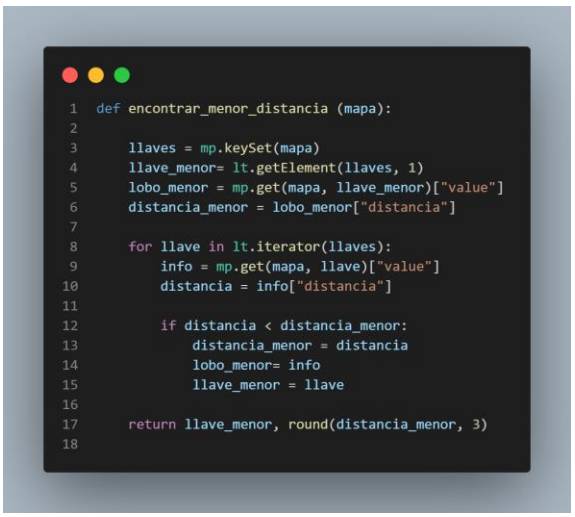
```python
def req_6(data_structs, fecha_inicial, fecha_final, sexo):
    """
    Función que soluciona el requerimiento 6
    """
    # TODO: Realizar el requerimiento 6

    fecha_inicial= fecha_inicial+ " 00:00"
    fecha_final= fecha_final + " 23:59"
    dicc_fecha_inicial = {"timestamp": fecha_inicial}
    dicc_fecha_final= {"timestamp": fecha_final}

    llave_lobos = mp.keySet(data_structs["lobos"])
    lista_lobos = lt.newList(datastructure="ARRAY_LIST")

    for llave in lt.iterator(llave_lobos):
        value= mp.get(data_structs["lobos"], llave)["value"]
        if value["animal-sex"] == sexo:
            lt.addLast(lista_lobos, llave)


    puntos_seguimiento = mp.newMap(numelements= 97 , maptype= "PROBING", loadfactor=0.5)
    puntos_seguimiento_lista = lt.newList("ARRAY_LIST")

    llaves_PS = mp.keySet(data_structs["ps"])

    for llave in lt.iterator(llaves_PS):
        if lt.isPresent(lista_lobos, llave) != 0:
            arbol= mp.get(data_structs["ps"], llave)["value"]
            llaves_arbol= om.keySet(arbol)
            for llave_arbol in lt.iterator(llaves_arbol):
                valor_arbol= om.get(arbol, llave_arbol)["value"]
                fila= valor_arbol["fila"]
                if (verificar_tiempo2(dicc_fecha_inicial, fila) == True) and (verificar_tiempo2(fila, dicc_fecha_final ) == True):
                    lt.addLast(puntos_seguimiento_lista, valor_arbol["ps"])
                    id= valor_arbol["fila"]["individual-local-identifier"]+ "_" + valor_arbol["fila"]["tag-local-identifier"]
                    if mp.contains(puntos_seguimiento, id) == False:
                        new_entry= lt.newList("ARRAY_LIST")
                        mp.put(puntos_seguimiento, id , new_entry)
                    else:
                        new_entry= mp.get(puntos_seguimiento, id)["value"]
                    lt.addLast(new_entry, valor_arbol["ps"])


    #mtp


    llaves_mtp= mp.keySet(data_structs["mtp"])
    mtps=lt.newList(datastructure="ARRAY_LIST")

    for llave in lt.iterator(llaves_mtp):
        value=mp.get(data_structs["mtp"], llave)["value"]
        lobos=lobos_diferentes_newgraph(value, puntos_seguimiento_lista)
        if lobos is not False:
            info={"mtp": llave, "pss": lobos}
            lt.addLast(mtps, info)

    grafo= gr.newGraph(datastructure= "ADJ_LIST", directed= True, size= 90000, cmpfunction=compareID)

    for ps in lt.iterator(puntos_seguimiento_lista):
        if not gr.containsVertex(grafo, ps):
            gr.insertVertex(grafo, ps)

    for mtp in lt.iterator(mtps):
        if not gr.containsVertex(grafo, mtp["mtp"]):
            gr.insertVertex(grafo, mtp["mtp"])


    addRouteConnections_PS_newgraph(data_structs, grafo,  puntos_seguimiento_lista)
    addRouteConnections_MTP_newgraph(grafo, mtps)

```

```python
#distancias
cantidad_lobos = lt.size(puntos_seguimiento_lista)
distancia_lobos = mp.newMap(numelements= cantidad_lobos*2+1, maptype= "PROBING", loadfactor=0.5)

arcos= gr.edges(grafo)

for arco in lt.iterator(arcos):
    lobo= dar_lobo_en_arco(arco)
    if mp.contains(distancia_lobos, lobo) == False:

        new_entry = {"id": lobo, "distancia": 0}
        mp.put(distancia_lobos, lobo, new_entry)
    else:
        new_entry = mp.get(distancia_lobos, lobo)["value"]
    new_entry["distancia"]+= e.weight(arco)


info_mayor = encontrar_mayor_distancia(distancia_lobos)
info_menor = encontrar_menor_distancia(distancia_lobos)

#mayor


distancia_mayor = info_mayor[1]
llave_mayor = info_mayor[0]
fila_mayor = mp.get(data_structs["lobos"], llave_mayor)["value"]

ps_mayor = mp.get(puntos_seguimiento, llave_mayor)["value"]

# ruta mas larga mayor
ruta_mayor = lt.newList("ARRAY_LIST")

primeros_mayores=lt.subList(ps_mayor, 1, 3)
ultimos_mayores=lt.subList(ps_mayor, lt.size(ps_mayor)-2, 3)

for ps in lt.iterator(primeros_mayores):
    lt.addLast(ruta_mayor, ps)

for ps in lt.iterator(ultimos_mayores):
    lt.addLast(ruta_mayor, ps)

arcos_mayor=lt.newList(datastructure="ARRAY_LIST")
for i in range(1, lt.size(ps_mayor)):
    ps1= lt.getElement(ps_mayor, i)
    ps2 = lt.getElement(ps_mayor, i+1)
    arco = gr.getEdge(grafo, ps1, ps2)
    lt.addLast(arcos_mayor, arco)

dicc_mayor={"id": llave_mayor, "fila":fila_mayor, "distancia": distancia_mayor,"nodos":lt.size(ps_mayor),"arcos":lt.size(arcos_mayor), "ruta":ruta_mayor }

#menor

distancia_menor = info_menor[1]
llave_menor = info_menor[0]
fila_menor = mp.get(data_structs["lobos"], llave_menor)["value"]

# ruta mas larga menor

ps_menor= mp.get(puntos_seguimiento, llave_menor)["value"]
ruta_menor = lt.newList("ARRAY_LIST")

primeros_menores=lt.subList(ps_menor, 1, 3)
ultimos_menores=lt.subList(ps_menor, lt.size(ps_menor)-2, 3)

for ps in lt.iterator(primeros_menores):
    lt.addLast(ruta_menor, ps)

for ps in lt.iterator(ultimos_menores):
    lt.addLast(ruta_menor, ps)

arcos_menor=lt.newList(datastructure="ARRAY_LIST")
for i in range(1, lt.size(ps_menor)):
    ps1= lt.getElement(ps_menor, i)
    ps2 = lt.getElement(ps_menor, i+1)
    arco = gr.getEdge(grafo, ps1, ps2)
    lt.addLast(arcos_menor, arco)

dicc_menor={"id": llave_menor, "fila": fila_menor, "distancia": distancia_menor, "nodos":lt.size(ps_menor),"arcos":lt.size(arcos_menor),"ruta":ruta_menor }

mapa=crear_mapa_folium()

trail1=[]
trail2 = []

for nodo in lt.iterator(ruta_mayor):
    crear_marcador(mapa, nodo)
    crear_ruta(trail1, nodo)

for nodo in lt.iterator(ruta_menor):
    crear_marcador(mapa, nodo)
    crear_ruta(trail2, nodo)

agregar_ruta(mapa, trail1)
agregar_ruta(mapa, trail2)

mapa.show_in_browser()


return dicc_mayor, dicc_menor

```

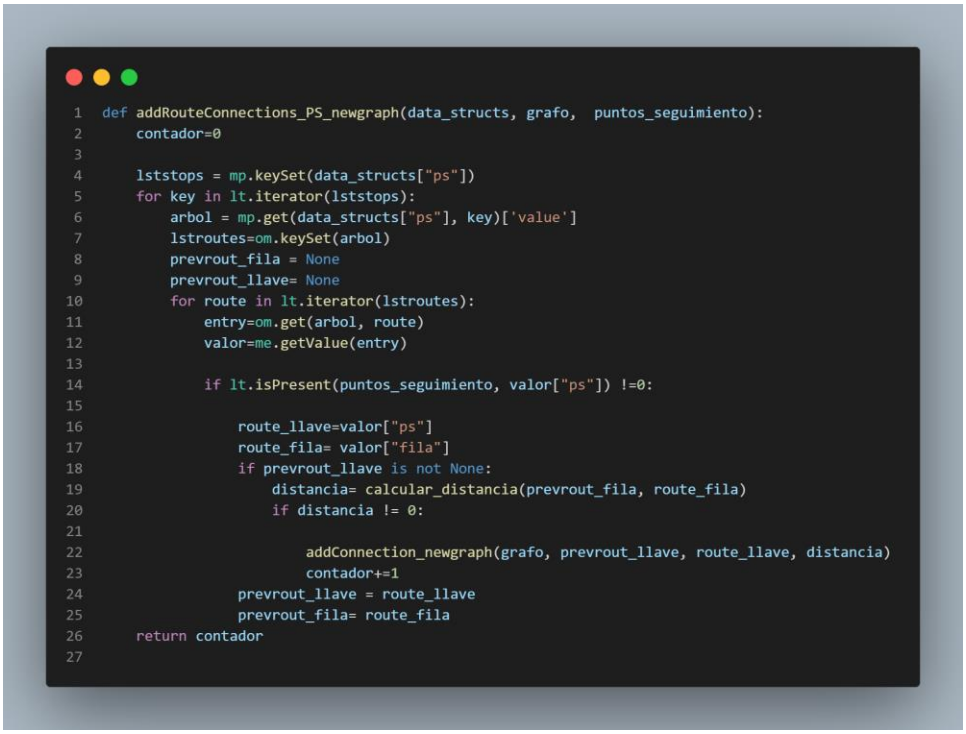


```python
def verificar_tiempo2 (info1, info2):

    info1_fc= info1["timestamp"]
    tiempo_info1= dt.strptime(info1_fc, "%Y-%m-%d %H:%M")
    info2_fc= info2["timestamp"]
    tiempo_info2= dt.strptime(info2_fc, "%Y-%m-%d %H:%M")


    if tiempo_info1 <= tiempo_info2:
        return True
    else:
        return False
```

```python
def encontrar_mayor_distancia (mapa):

    llaves = mp.keySet(mapa)
    llave_mayor= lt.getElement(llaves, 1)
    lobo_mayor = mp.get(mapa, llave_mayor)["value"]
    distancia_mayor = lobo_mayor["distancia"]

    for llave in lt.iterator(llaves):
        info = mp.get(mapa, llave)["value"]
        distancia = info["distancia"]

        if distancia > distancia_mayor:
            distancia_mayor = distancia
            lobo_mayor= info
            llave_mayor = llave

    return llave_mayor, round(distancia_mayor, 3)
```



```python
def encontrar_menor_distancia (mapa):

    llaves = mp.keySet(mapa)
    llave_menor= lt.getElement(llaves, 1)
    lobo_menor = mp.get(mapa, llave_menor)["value"]
    distancia_menor = lobo_menor["distancia"]

    for llave in lt.iterator(llaves):
        info = mp.get(mapa, llave)["value"]
        distancia = info["distancia"]

        if distancia < distancia_menor:
            distancia_menor = distancia
            lobo_menor= info
            llave_menor = llave

    return llave_menor, round(distancia_menor, 3)

```



```python
def addRouteConnections_PS_newgraph(data_structs, grafo,  puntos_seguimiento):
    contador=0

    lststops = mp.keySet(data_structs["ps"])
    for key in lt.iterator(lststops):
        arbol = mp.get(data_structs["ps"], key)['value']
        lstroutes=om.keySet(arbol)
        prevrout_fila = None
        prevrout_llave= None
        for route in lt.iterator(lstroutes):
            entry=om.get(arbol, route)
            valor=me.getValue(entry)

            if lt.isPresent(puntos_seguimiento, valor["ps"]) !=0:

                route_llave=valor["ps"]
                route_fila= valor["fila"]
                if prevrout_llave is not None:
                    distancia= calcular_distancia(prevrout_fila, route_fila)
                    if distancia != 0:

                        addConnection_newgraph(grafo, prevrout_llave, route_llave, distancia)
                        contador+=1
                prevrout_llave = route_llave
                prevrout_fila= route_fila
    return contador

```

```python
def addRouteConnections_MTP_newgraph(grafo, mtps):
    contador=0
    for mtp in lt.iterator(mtps):
        for ps in lt.iterator(mtp["pss"]):
            if gr.containsVertex(grafo, mtp["mtp"]):
                    addConnection_newgraph(grafo, mtp["mtp"] , ps, 0)
                    addConnection_newgraph(grafo, ps , mtp["mtp"], 0)
                    contador+=2
    return contador

```

Descripción

Este requerimiento se encarga de identificar diferencias en los corredores migratorios según el tipo de individuo en un rango de fechas

Entrada	Data_structs, fecha_inicial, fecha_final, sexo
Salidas	Un diccionario con la información del individuo que recorrió más distancia con su ruta más larga. Un diccionario con la información del individuo que recorrió menor distancia con su ruta más larga.
Implementado (Sí/No)	Si, grupal

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1: Se buscan los lobos que tengan el sexo dado y se agrega a una lista	O(1) porque solo hay 46 lobos
Paso 2: Se recorren las llaves de los puntos de seguimiento (los lobos) y se verifica si este es de los lobos ya encontrados con el mismo sexo.	O(n log n) donde n es la cantidad de puntos de seguimiento en cada lobo
Paso 3: Por cada punto de seguimiento se verifica si está dentro en el rango de tiempo y lo agrega a un mapa y a una lista.	O(k), donde k son los puntos de seguimiento de los lobos que tienen el mismo sexo.
Paso 4: Se recorren los mtp y con una función auxiliar se verifica si sigue existen cada uno según la condición de que pasen dos lobos o más.	O(v), donde v son la cantidad de mtp
Paso 5: Se crea el grafo con los ps y mtp encontrados y se hacen las conexiones	O (log m + m), donde m es la cantidad de ps que pertenecen a un lobo especifico.
Paso 6: se crea un mapa donde se van a guardar las distancias que recorre cada lobo, con un diccionario con el id del lobo y la distancia que recorrió	O(1)
Paso 7: se recorren los arcos del grafo. Con una función auxiliar se encuentra el lobo en el arco y verifica si está en el mapa. En el caso que no crea el diccionario. En el caso que si este suma la distancia.	O(a), donde a son los arcos del nuevo grafo creado
Paso 8: Se encuentra el lobo que recorrió la mayor distancia y el lobo que recorrió la menor distancia con funciones auxiliares.	O(n), n siendo la cantidad de lobos que hay en el nuevo grafo
Paso 9: Se saca la fila correspondiente al lobo que más distancia recorrió y los puntos de seguimiento correspondientes al lobo	O(1)
Paso 10: Se sacan los 3 primeros y 3 últimos nodos de los puntos de seguimiento que corresponden a los puntos del camino más largo	O(1)
Paso 11: Se sacan los arcos del camino a partir de los nodos del camino	O(m) donde m es la cantidad de nodos en el camino (m < v)
Paso 12: Se agrega la información del mayor a un diccionario para retornarlo	O(1)
Pasos 13-16: Se repiten los pasos 9 a 12 para encontrar la información del lobo que menos distancia recorrió y retornarla	O(m)
TOTAL	***O(log n)***

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).

	Máquina 1	Máquina 2
Procesadores	12th Gen Intel(R) Core(TM) i7-1255U 1.70 GHz	AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx 2.10 GHz
Memoria RAM (GB)	16,0 GB (15,7 GB utilizable)	16.0 GB (13.9 GB usable)
Sistema Operativo	Sistema operativo de 64 bits, procesador x64, Windows 11 Home Single Language	Windows 11

Máquina 1:

Entrada	Datos entrada	Tiempo (ms)
small	66	614,95
5 pct	468	2746,01
10 pct	934	7254,19
20 pct	1821	38701,6
30 pct	2705	35852,8
50 pct	4636	89144
80 pct	7645	185465
large	9642	289122

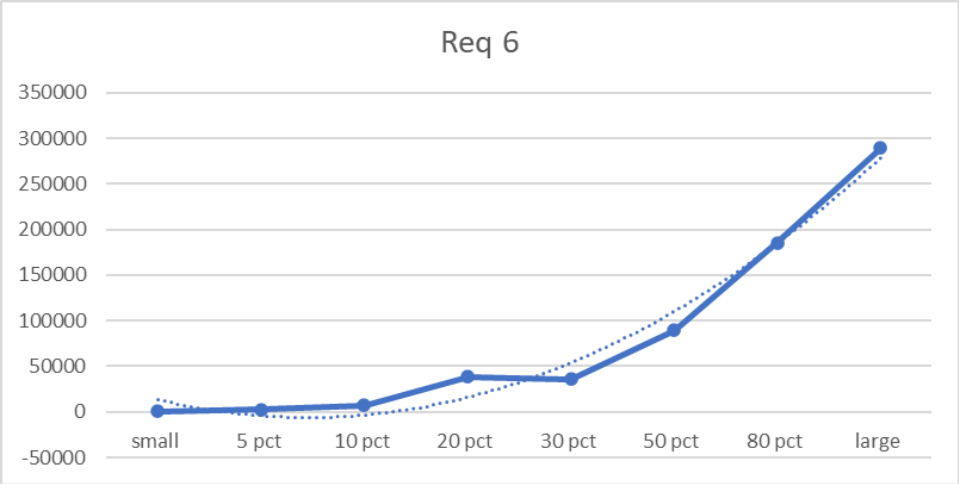
Máquina 2:

Entrada	Datos entrada	Tiempo (ms)
small	66	1054.16
5 pct	468	11241.5
10 pct	934	21494.5
20 pct	1821	55368
30 pct	2705	77056.1
50 pct	4636	181535
80 pct	7645	370591
large	9642	598854

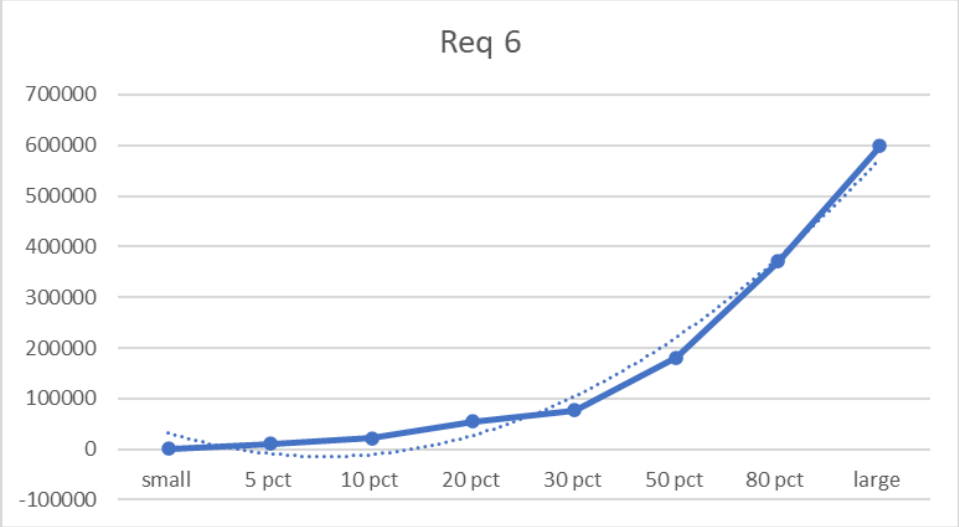
Gráficas

Las gráficas con la representación de las pruebas realizadas.

Máquina 1:



Máquina 2:



Análisis

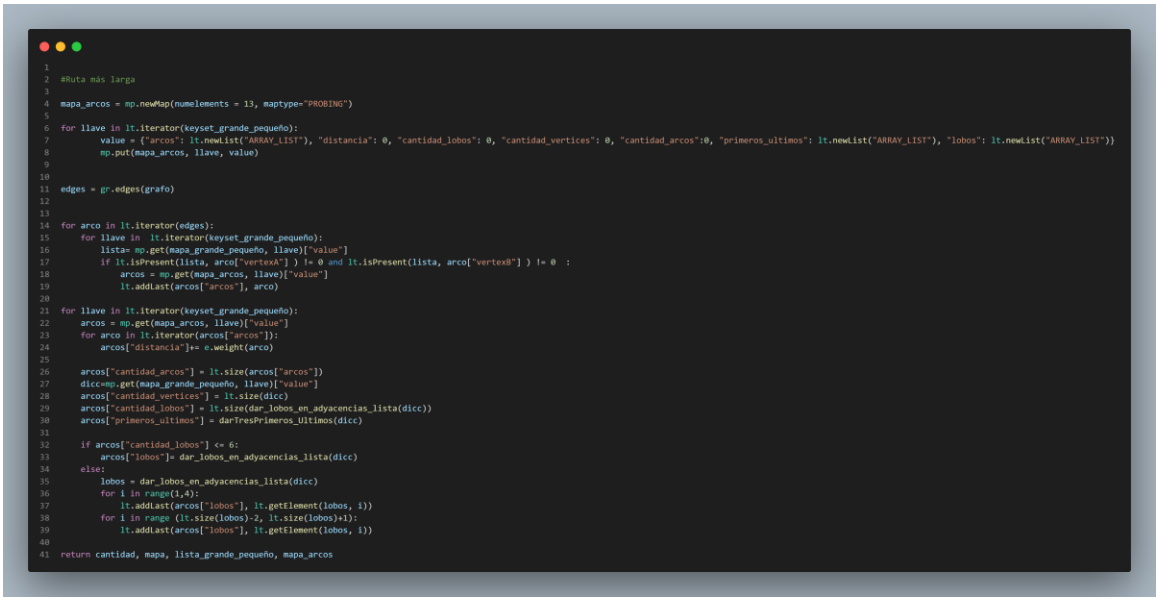
Este requerimiento tiene una complejidad de O(log n) porque se recorre el índice data_structs[“ps”] que guarda la información en un mapa que contiene árboles y sacar la información de un árbol tiene una complejidad de O(log n) y se hacer esta operación n veces, entonces O(n log n). Las gráficas de tiempo muestran un comportamiento que puede ser explicado a partir de esta complejidad, pues las curvaturas coinciden.

Requerimiento <<7>>

```python
def req_7(data_structs, fecha_inicial, fecha_final, temp_min, temp_max):
    """
    Función que soluciona el requerimiento 7
    """
    # TODO: Realizar el requerimiento 7

    fecha_inicial= fecha_inicial+ " 00:00"
    fecha_final= fecha_final + " 23:59"
    dicc_fecha_inicial = {"timestamp": fecha_inicial}
    dicc_fecha_final= {"timestamp": fecha_final}

    puntos_seguimiento = mp.newMap(numelements= 97 , maptype= "PROBING", loadfactor=0.5)
    puntos_seguimiento_lista = lt.newList("ARRAY_LIST")

    llaves_PS = mp.keySet(data_structs["ps"])

    for llave in lt.iterator(llaves_PS):
        arbol= mp.get(data_structs["ps"], llave)["value"]
        llaves_arbol= om.keySet(arbol)
        for llave_arbol in lt.iterator(llaves_arbol):
            valor_arbol= om.get(arbol, llave_arbol)["value"]
            fila= valor_arbol["fila"]
            if (verificar_tiempo2(dicc_fecha_inicial, fila) == True) and (verificar_tiempo2(fila, dicc_fecha_final ) == True) and (float(temp_min)<=float(fila["external-temperature"])) and (float(temp_max)>=float(fila["external-temperature"])):
                if lt.isPresent(puntos_seguimiento_lista, valor_arbol["ps"])== 0:
                    lt.addLast(puntos_seguimiento_lista, valor_arbol["ps"])
                id= valor_arbol["fila"]["individual-local-identifier"]+ "_" + valor_arbol["fila"]["tag-local-identifier"]
                if mp.contains(puntos_seguimiento, id) == False:
                    new_entry= lt.newList("ARRAY_LIST")
                    mp.put(puntos_seguimiento, id , new_entry)
                else:
                    new_entry= mp.get(puntos_seguimiento, id)["value"]
                lt.addLast(new_entry, valor_arbol["ps"])


    #mtp
    llaves_mtp= mp.keySet(data_structs["mtp"])
    mtps=lt.newList(datastructure="ARRAY_LIST")

    for llave in lt.iterator(llaves_mtp):
        value=mp.get(data_structs["mtp"], llave)["value"]
        lobos=lobos_diferentes_newgraph(value, puntos_seguimiento_lista)
        if lobos is not False:
            info={"mtp": llave, "pss": lobos}
            lt.addLast(mtps, info)


    grafo= gr.newGraph(datastructure= "ADJ_LIST", directed= True, size= 90000, cmpfunction=compareID)

    for ps in lt.iterator(puntos_seguimiento_lista):
        if not gr.containsVertex(grafo, ps):
            gr.insertVertex(grafo, ps)

    for mtp in lt.iterator(mtps):
        if not gr.containsVertex(grafo, mtp["mtp"]):
            gr.insertVertex(grafo, mtp["mtp"])


    addRouteConnections_PS_newgraph(data_structs, grafo,  puntos_seguimiento_lista)
    addRouteConnections_MTP_newgraph(grafo, mtps)

```

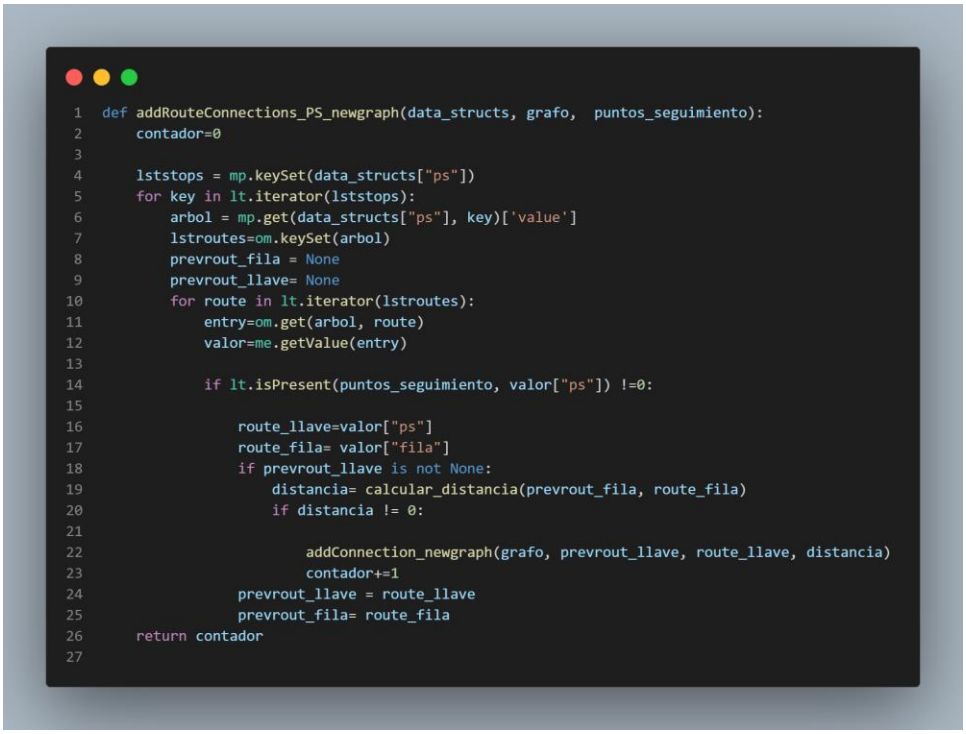

```python
def req_7(data_structs, fecha_inicial, fecha_final, temp_min, temp_max):
    """
    Función que soluciona el requerimiento 7
    """
    # TODO: Realizar el requerimiento 7

    fecha_inicial= fecha_inicial+ " 00:00"
    fecha_final= fecha_final + " 23:59"
    dicc_fecha_inicial = {"timestamp": fecha_inicial}
    dicc_fecha_final= {"timestamp": fecha_final}

    puntos_seguimiento = mp.newMap(numelements= 97 , maptype= "PROBING", loadfactor=0.5)
    puntos_seguimiento_lista = lt.newList("ARRAY_LIST")

    llaves_PS = mp.keySet(data_structs["ps"])

    for llave in lt.iterator(llaves_PS):
        arbol= mp.get(data_structs["ps"], llave)["value"]
        llaves_arbol= om.keySet(arbol)
        for llave_arbol in lt.iterator(llaves_arbol):
            valor_arbol= om.get(arbol, llave_arbol)["value"]
            fila= valor_arbol["fila"]
            if (verificar_tiempo2(dicc_fecha_inicial, fila) == True) and (verificar_tiempo2(fila, dicc_fecha_final ) == True) and (float(temp_min)<=float(fila["external-temperature"])) and (float(temp_max)>=float(fila["external-temperature"])):
                if lt.isPresent(puntos_seguimiento_lista, valor_arbol["ps"])== 0:
                    lt.addLast(puntos_seguimiento_lista, valor_arbol["ps"])
                id= valor_arbol["fila"]["individual-local-identifier"]+ "_" + valor_arbol["fila"]["tag-local-identifier"]
                if mp.contains(puntos_seguimiento, id) == False:
                    new_entry= lt.newList("ARRAY_LIST")
                    mp.put(puntos_seguimiento, id , new_entry)
                else:
                    new_entry= mp.get(puntos_seguimiento, id)["value"]
                lt.addLast(new_entry, valor_arbol["ps"])


    #mtp
    llaves_mtp= mp.keySet(data_structs["mtp"])
    mtps=lt.newList(datastructure="ARRAY_LIST")

    for llave in lt.iterator(llaves_mtp):
        value=mp.get(data_structs["mtp"], llave)["value"]
        lobos=lobos_diferentes_newgraph(value, puntos_seguimiento_lista)
        if lobos is not False:
            info={"mtp": llave, "pss": lobos}
            lt.addLast(mtps, info)


    grafo= gr.newGraph(datastructure= "ADJ_LIST", directed= True, size= 90000, cmpfunction=compareID)

    for ps in lt.iterator(puntos_seguimiento_lista):
        if not gr.containsVertex(grafo, ps):
            gr.insertVertex(grafo, ps)

    for mtp in lt.iterator(mtps):
        if not gr.containsVertex(grafo, mtp["mtp"]):
            gr.insertVertex(grafo, mtp["mtp"])


    addRouteConnections_PS_newgraph(data_structs, grafo,  puntos_seguimiento_lista)
    addRouteConnections_MTP_newgraph(grafo, mtps)

```

```python
keyset_grande_pequeño = mp.keySet(mapa_grande_pequeño)

mapa = mp.newMap(numelements= 13, maptype="PROBING", loadfactor=0.5)


for llave in lt.iterator(keyset_grande_pequeño):

    diccionario = {}
    mp.put(mapa, llave, diccionario)

#encontrar 3 primeros y 3 ultimos

for llave in lt.iterator(keyset_grande_pequeño):
    primeros_ultimos = lt.newList("ARRAY_LIST")
    value= mp.get(mapa_grande_pequeño, llave)["value"]
    if lt.size(value)<=6:
        for dato in lt.iterator(value):
            lt.addLast(primeros_ultimos, dato)

    else:
        for i in range (1,4):
            fila= lt.getElement(value, i)
            lt.addLast(primeros_ultimos,  fila)
        for i in range (lt.size(value)-2, lt.size(value)+1):
            fila= lt.getElement(value, i)
            lt.addLast(primeros_ultimos,  fila)


    dicc=mp.get(mapa, llave)["value"]
    dicc["primeros_ultimos"]= primeros_ultimos

#tamaño

for llave in lt.iterator(keyset_grande_pequeño):
    dicc = mp.get(mapa, llave)["value"]
    value= mp.get(mapa_grande_pequeño, llave)["value"]
    dicc["tamaño"]= lt.size(value)


#encontrar lat y long

for llave in lt.iterator(keyset_grande_pequeño):

    value= mp.get(mapa_grande_pequeño, llave)["value"]
    mayor = buscar_mayor_long_lat_lista(value)
    menor = buscar_menor_long_lat_lista (value)
    lat_max  = lt.getElement(mayor, 2)
    long_max = lt.getElement(mayor, 1)
    lat_min = lt.getElement(menor, 2)
    long_min = lt.getElement(menor, 1)

    dicc = mp.get(mapa, llave)["value"]
    dicc["min_lat"] = lat_min
    dicc["max_lat"] = lat_max
    dicc["min_lon"] = long_min
    dicc["max_lon"] = long_max


#encontrar lobos
for llave in lt.iterator(keyset_grande_pequeño):

    lista_lobos = lt.newList(datastructure= "ARRAY_LIST")
    value= mp.get(mapa_grande_pequeño, llave)["value"]
    lobos = dar_lobos_en_adyacencias_lista(value)
    cantidad_lobos = lt.size(lobos)


    if cantidad_lobos <=6:
        for lobo in lt.iterator(lobos):
            fila= mp.get(data_structs["lobos"], lobo)["value"]
            lt.addLast(lista_lobos,fila )

    else:
        for i in range (1,4):
            lobo= lt.getElement(lobos, i)
            fila= mp.get(data_structs["lobos"], lobo)["value"]
            lt.addLast(lista_lobos, fila )

        for i in range (lt.size(lobos)-2, lt.size(lobos)+1):
            lobo= lt.getElement(lobos, i)
            fila= mp.get(data_structs["lobos"], lobo)["value"]
            lt.addLast(lista_lobos, fila )


    dicc=mp.get(mapa, llave)["value"]
    dicc["cantidad_lobos"] = cantidad_lobos
    dicc["informacion_lobos"]= lista_lobos

```



```
#Ruta más larga

mapa_arcos = mp.newMap(numelements = 13, maptype="PROBING")

for llave in lt.iterator(keyset_grande_pequeño):
        value = {"arcos": lt.newList("ARRAY_LIST"), "distancia": 0, "cantidad_lobos": 0, "cantidad_vertices": 0, "cantidad_arcos":0, "primeros_ultimos": lt.newList("ARRAY_LIST"), "lobos": lt.newList("ARRAY_LIST")}
        mp.put(mapa_arcos, llave, value)


edges = gr.edges(grafo)


for arco in lt.iterator(edges):
    for llave in  lt.iterator(keyset_grande_pequeño):
        lista= mp.get(mapa_grande_pequeño, llave)["value"]
        if lt.isPresent(lista, arco["vertexA"] ) != 0 and lt.isPresent(lista, arco["vertexB"] ) != 0  :
            arcos = mp.get(mapa_arcos, llave)["value"]
            lt.addLast(arcos["arcos"], arco)

for llave in lt.iterator(keyset_grande_pequeño):
    arcos = mp.get(mapa_arcos, llave)["value"]
    for arco in lt.iterator(arcos["arcos"]):
        arcos["distancia"]+= e.weight(arco)

    arcos["cantidad_arcos"] = lt.size(arcos["arcos"])
    dicc=mp.get(mapa_grande_pequeño, llave)["value"]
    arcos["cantidad_vertices"] = lt.size(dicc)
    arcos["cantidad_lobos"] = lt.size(dar_lobos_en_adyacencias_lista(dicc))
    arcos["primeros_ultimos"] = darTresPrimeros_Ultimos(dicc)

    if arcos["cantidad_lobos"] <= 6:
        arcos["lobos"]= dar_lobos_en_adyacencias_lista(dicc)
    else:
        lobos = dar_lobos_en_adyacencias_lista(dicc)
        for i in range(1,4):
            lt.addLast(arcos["lobos"], lt.getElement(lobos, i))
        for i in range (lt.size(lobos)-2, lt.size(lobos)+1):
            lt.addLast(arcos["lobos"], lt.getElement(lobos, i))

return cantidad, mapa, lista_grande_pequeño, mapa_arcos
```

```
def lobos_diferentes_newgraph(lista, puntos_seguimiento):

    lista_lobos= lt.newList(datastructure= "ARRAY_LIST")
    lista_ps = lt.newList(datastructure= "ARRAY_LIST")


    for lobo in lt.iterator(lista):
        ps= lobo["ps"]
        info_id= separar_id_completo(ps)
        id = info_id[2]+ "_"+ info_id[3]
        if lt.isPresent(lista_lobos, id) == 0 and lt.isPresent(puntos_seguimiento, ps)!=0:
            lt.addLast(lista_ps, ps)
            lt.addLast(lista_lobos, id)

    if lt.size(lista_lobos)>= 2:
        return lista_ps
    else:
        return False
```



```python
def addRouteConnections_PS_newgraph(data_structs, grafo,  puntos_seguimiento):
    contador=0

    lststops = mp.keySet(data_structs["ps"])
    for key in lt.iterator(lststops):
        arbol = mp.get(data_structs["ps"], key)['value']
        lstroutes=om.keySet(arbol)
        prevrout_fila = None
        prevrout_llave= None
        for route in lt.iterator(lstroutes):
            entry=om.get(arbol, route)
            valor=me.getValue(entry)

            if lt.isPresent(puntos_seguimiento, valor["ps"]) !=0:

                route_llave=valor["ps"]
                route_fila= valor["fila"]
                if prevrout_llave is not None:
                    distancia= calcular_distancia(prevrout_fila, route_fila)
                    if distancia != 0:

                        addConnection_newgraph(grafo, prevrout_llave, route_llave, distancia)
                        contador+=1
                prevrout_llave = route_llave
                prevrout_fila= route_fila
    return contador

```

```python
def addRouteConnections_MTP_newgraph(grafo, mtps):
    contador=0
    for mtp in lt.iterator(mtps):
        for ps in lt.iterator(mtp["pss"]):
            if gr.containsVertex(grafo, mtp["mtp"]):
                    addConnection_newgraph(grafo, mtp["mtp"] , ps, 0)
                    addConnection_newgraph(grafo, ps , mtp["mtp"], 0)
                    contador+=2
    return contador

```

Descripción

Este requerimiento se encarga de identificar cambios en el territorio de las manadas (cfc) según condiciones climáticas en un rango de tiempo.

Entrada	data_structs, fecha inicial, fecha final, temperatura inicial, temperatura final
Salidas	Mapa con la información de las 3 manadas más grandes y las 3 manadas más pequeñas, lista con el orden de los cfc, cantidad de cfc total, mapa con la información del recorrido más largo

Implementado (Sí/No)	Si, grupal

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1: Se buscan los lobos que tengan el sexo dado y se agrega a una lista	O(n), n siendo la cantidad de lobos que hay
Paso 2: Por cada punto de seguimiento se verifica si está dentro en el rango de tiempo y de temperatura para agregarlo a un mapa y a una lista.	O(V)
Paso 3: Se recorren los mtp y con una función auxiliar se verifica si sigue existen cada uno según la condición de que pasen dos lobos o más.	O(v), donde v son la cantidad de mtp
Paso 4: Se crea el grafo con los ps y mtp encontrados y se hacen las conexiones	O (m log m), donde m es la cantidad de ps que pertenecen a un lobo especifico.
Paso 5: Se hace un search de Kosaraju para encontrar los componentes conectados del grafo	O(V + A)
Paso 6: Se encuentra la cantidad de cfc	O(1)
Paso 7: Se sacan las llaves del mapa de ps y se saca una llave de referencia.	O(V+k)
Paso 8: Con la llave de referencia y los cfc ya encontrados se saca el mapa con los vértices de cada cfc.	O(V)
Paso 9: Se saca el keyset de los vertices de los cfc	O(V+k)
Paso 10: En un mapa se va agregando el id del cfc como llave y su valor como los vértices que pertenecen a dicho cfc.	O(V)
Paso 11: Se encuentran las 3 manadas más grandes y las 3 manadas más pequeñas (con mayor cantidad de vertices) y se agregan a un mapa	O(1)
Paso 12: Paso 8: Se crea un mapa donde la llave va a ser el id del cfc y su valor un diccionario donde se va a almacenar la información del cfc.	O(1)
Paso 13: Se encuentran los 3 primeros y últimos puntos de encuentro de las 3 manadas más grandes y las 3 manadas más pequeñas	O(1)
Paso 14: Se encuentra el tamaño 3 manadas más grandes y las 3 manadas más pequeñas	O(1)
Paso 15: Busca la latitud y longitud minina y máxima de las 3 manadas más grandes y las 3 manadas más pequeñas con funciones auxiliares.	O(e) donde e es la cantidad de vértices que hay en cada componente fuertemente conectado.
Paso 16: Encontrar los tres primeros y últimos lobos de las 3 manadas más grandes y las 3 manadas más pequeñas.	O(e) donde e es la cantidad de vértices que hay en cada componente fuertemente conectado.

Paso 17: Se crea un nuevo mapa donde se van a guardar los arcos del camino más largo y la información correspondiente (cantidad arcos, vértices, lobos, 3 primeros y últimos vértices, etc.)	O(1)
Paso 18: se sacan todos los arcos en el nuevo grafo construido	O(a), donde a es la cantidad de arcos en el nuevo grafo a<A
Paso 19: Se recorren todos los arcos y los nodos de las 3 manadas más grandes y las 3 manadas más pequeñas. Se verifica que los uno de los dos vértices del arco este dentro de los nodos. En el caso de que estén se agregan al mapa.	O(a)
Paso 20: Se recorren los arcos del nuevo mapa y se va agregando el peso (distancia de un punto al otro) para conseguir la distancia de la ruta más larga	O(a)
Paso 21: Se saca la información de cantidad de arcos, cantidad de vértices, cantidad de lobos, los tres primeros y tres últimos lobos distintos y se saca los tres primero y tres últimos nodos del camino con una función auxiliar	O(e)
TOTAL	***O(m log m)***

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).

	Máquina 1	Máquina 2
Procesadores	12th Gen Intel(R) Core(TM) i7-1255U 1.70 GHz	AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx 2.10 GHz
Memoria RAM (GB)	16,0 GB (15,7 GB utilizable)	16.0 GB (13.9 GB usable)
Sistema Operativo	Sistema operativo de 64 bits, procesador x64, Windows 11 Home Single Language	Windows 11

Máquina 1:

Entrada	Datos entrada	Tiempo (ms)
small	66	820,897

5 pct	468	5057,73
10 pct	934	14033,9
20 pct	1821	73828,2
30 pct	2705	62738
50 pct	4636	140158
80 pct	7645	264424
large	9642	397455

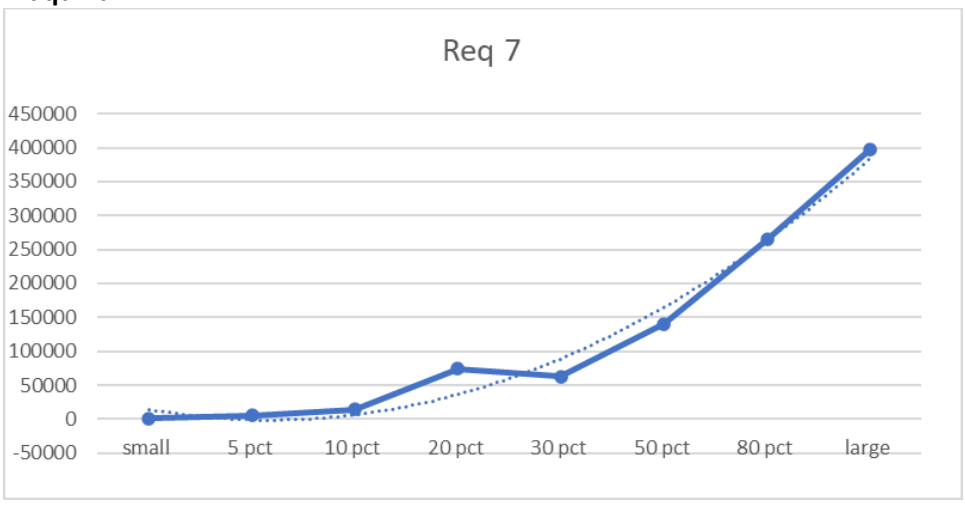
Máquina 2:

Entrada	Datos entrada	Tiempo (ms)
small	66	2026.16
5 pct	468	14054.6
10 pct	934	38508.8
20 pct	1821	73351
30 pct	2705	113216
50 pct	4636	132098
80 pct	7645	249112
large	9642	513303

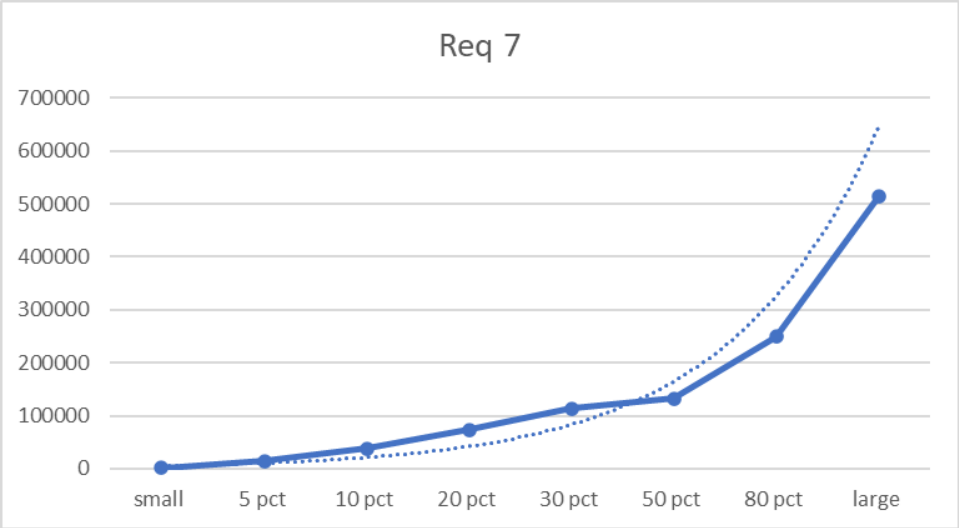
Gráficas

Las gráficas con la representación de las pruebas realizadas.

Máquina 1:



Máquina 2:



Análisis

Este requerimiento tiene una complejidad de O(m log m) donde m es la cantidad de puntos de seguimiento que pertenecen a cada lobo. Esto debido a que al crear el nuevo grafo se recorre el índice data_structs[“ps”] que está compuesto por árboles dentro de mapas. Como se hace m veces y sacar información de un árbol tiene una complejidad de O(log m) entonces se llega a la complejidad descrita.

A pesar de que se hace un Kosaraju con el grafo construido para sacar los componentes fuertemente conectados, como el grafo construido normalmente tiene menos vértices que el original la complejidad de este, que sería O(V + A) va a ser menor a O (m log m) . Las gráficas muestran un comportamiento que se acerca al deseado porque coinciden en la curvatura.