Universidad de los Andes
Facultad de Ingeniería

Departamento de Ingeniería de Sistemas y Computación
Estructuras de Datos y Algoritmos
ISIS-1225

ABET Engineering Accreditation Commission

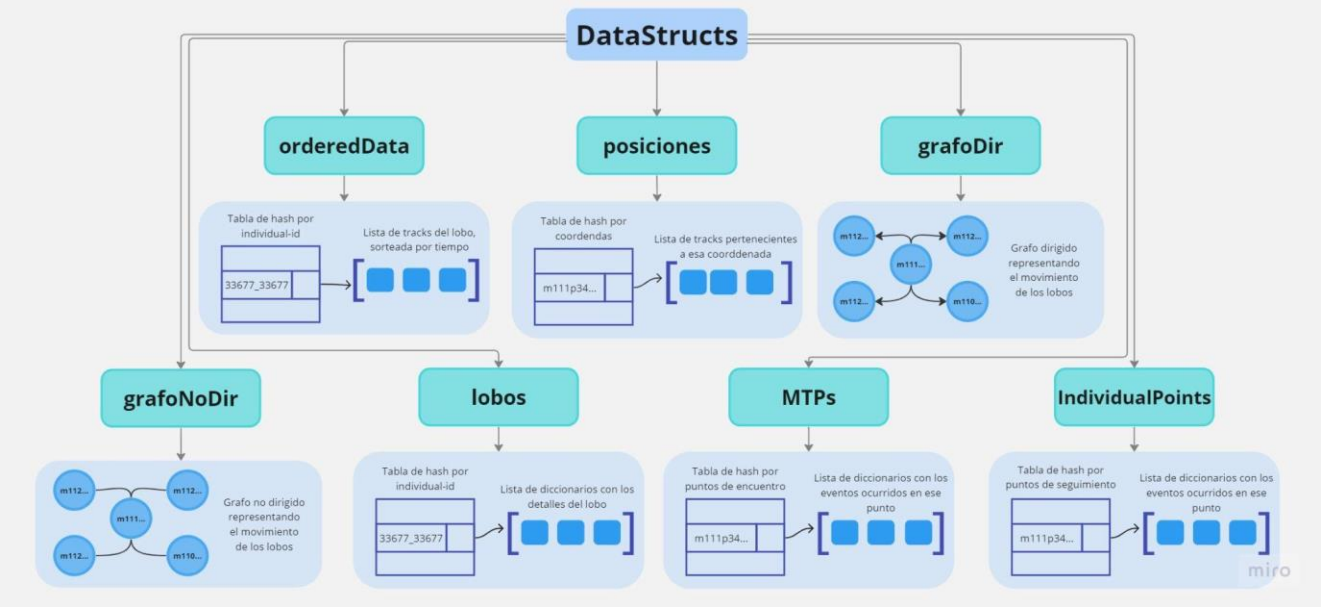
ANÁLISIS DEL RETO

Jacobo Zarruk, 202223913, j.zarruk@uniandes.edu.co

Angélica Ortiz,202222480, a.ortizb2@uniandes.edu.co

María José Amorocho, 202220179, m.amorocho@uniandes.edu.co

Carga de datos



Descripción

Para la carga de datos se presentan siete índices: data_structs['orderedData'], un mapa que guarda como llave el individual-id de cada lobo, y como valor, una lista diccionarios con los eventos de cada lobo sorteada por tiempo; data_structs['posiciones'], un mapa que tiene como llave la posición (ya formateada, pero sin el individual-id) de un evento, y como valor, una lista de diccionarios con los eventos que allí ocurrieron, siempre y cuando se trate de un lobo diferente; data_structs['grafoDir'], un grafo dirigido representando el movimiento de los lobos; data_structs['grafoNoDir'], un grafo no dirigido con el movimiento de los individuos; data_structs['lobos'], un mapa que tiene como llaves el individual-id de cada lobo, y como valor un diccionario con los detalles del individuo; data_structs['MTPs'], un mapa que tiene como llave la coordenada de un punto de encuentro, y como valor una lista de diccionarios de eventos que corresponden a ese punto de encuentro, y data_structs['individualPoints'], un mapa con los puntos individuales de cada lobo; como llave tiene la posición del evento y como valor una lista de diccionarios con los eventos allí ocurridos.

La carga de datos se hace con los dos archivos provisionados: Sobre cada uno se itera para leer los datos y construir los mapas y grafos anteriormente mencionados; así pues, la complejidad en de O(n.)

Requerimiento 1

Objetivo: conocer si existe un camino utilizado entre dos puntos de encuentro para lobos

Descripción

```
def req_1(data_structs,initialPoint,destPoint):
    """
    Función que soluciona el requerimiento 1
    """
    # TO DO: Realizar el requerimiento 1
    data_structs['search']= dfs.DepthFirstSearch(data_structs['grafoDir'],initialPoint)
    camino = dfs.hasPathTo(data_structs['search'],destPoint)
    lstCamino = lt.newList('ARRAY_LIST')
    trackingPoints = 0
    if camino:
        ruta = dfs.pathTo(data_structs['search'],destPoint)
        while (not st.isEmpty(ruta)):
            lt.addLast(lstCamino,st.pop(ruta))

    if camino == False:
        return 0,0,[]
```

```
else:
    lastElem = lt.lastElement(lstCamino)
    lt.addLast(lstCamino,lastElem)
    totalDist = 0
    lasttrack = None
    totalMtps = 0
    lstReturn = lt.newList('ARRAY_LIST')
    for nodo in lt.iterator(lstCamino):
        lstInfo = lt.newList('ARRAY_LIST')
        if lasttrack != None:
            try:
                totalDist += gr.getEdge(data_structs['grafoDir'],lasttrack,nodo)['weight']
            except:
                totalDist += 0
            entry = mp.get(data_structs['individualPoints'],lasttrack)
            if entry != None:
                commonWolfs = 1
                wolfsId = me.getValue(entry)['elements'][0]['individual-id']
                nodeId = me.getValue(entry)['elements'][0]['node-id']
                trackingPoints += 1
            else:
                entry = mp.get(data_structs['MTPs'],lasttrack)
                totalMtps += 1
                value = me.getValue(entry)
                nodeId = lasttrack
                commonWolfs = lt.size(value)
                wolfsIds = lt.newList('ARRAY_LIST')
                for event in lt.iterator(value):
                    lt.addLast(wolfsIds,event['individual-id'])
                if lt.size(wolfsIds) > 6:
                    wolfsId = getIFirstandLast(wolfsIds,3)
                else:
                    res = ""
                    for wolf in lt.iterator(wolfsIds):
                        res += wolf+", "
                    wolfsId = res
            value = me.getValue(entry)['elements'][0]
            lt.addLast(lstInfo,nodeId)
            lt.addLast(lstInfo,value['location-long'])
            lt.addLast(lstInfo,value['location-lat'])
            lt.addLast(lstInfo,commonWolfs)
            lt.addLast(lstInfo,wolfsId)
            if lasttrack != nodo:
                lt.addLast(lstInfo,nodo)
            else:
                lt.addLast(lstInfo,'Unknown')
            try:
                lt.addLast(lstInfo,(gr.getEdge(data_structs['grafoNoDir'],lasttrack,nodo)['weight']))
            except:
                lt.addLast(lstInfo,0)
            lt.addLast(lstReturn,lstInfo)
        lasttrack = nodo

    if lt.size(lstReturn) > 10:
        rta = getiFirstandLast(lstReturn,5)
    else:
        rta = lstReturn
    totalNodes = lt.size(lstCamino)
    return totalNodes-1,trackingPoints,totalDist,totalMtps,rta
```

En la primera parte, se hace un dfs en el grafo dirigido dado un punto de inicio, y se calcula si hay un camino que conecte a este vértice con el vértice de destino. Si no existe una ruta, se retorna (0,0,[]); de lo contrario, se analiza la lista de vértices que conforman el camino para obtener a partir de estos la información solicitada (distancia total, puntos de encuentro, nodos de seguimiento, cinco primeros y cinco últimos vértices que definen la ruta)

Entrada	Identificador del punto de encuentro de origen, identificador del punto de encuentro de destino
Salidas	Distancia total del camino, total de puntos de encuentro en el camino, total de nodos de seguimiento en el camino, cinco primeros y cinco últimos vértices del camino con su respectiva información.
Implementado (Sí/No)	Sí. Implementado por María José

Análisis de complejidad

Pasos	Complejidad
Se define el índice data_structs[‘search’] como una estructura para reconocer los vértices conectados al punto de partida	O(1)
A data_structs[‘search’] se le hace un recorrido dfs para saber si hay o no una ruta que conecte al punto de inicio con el punto de destino, y cuál es esta ruta.	O(V + E)
Si hay camino, se calcula la ruta a seguir desde el punto de inicio al punto final. Cada vértice que conforma está a esta ruta se agrega a la lista ‘lstCamino’.	O(n)
Si no hay camino, se retorna (0,0, [])	O(1)
Cuando existe la ruta, a ‘lstCamino’ se le agrega nuevamente su último elemento; esto para obtener la información del último nodo al final del algoritmo.	O(1)
Para cada vértice que exista en ‘lstCamino’ se toman dos vértices consecutivos en la lista; el primero será el vértice de inicio y el segundo el vértice de destino.	O(n)
Se toma el arco que conecta al vértice de inicio con el vértice de destino y se suma su peso a la distancia total recorrida (‘totalDist’)	
Se determina si el vértice de inicio corresponde a un punto de encuentro (MTP) o un punto de seguimiento. Para esto se busca primero en el mapa data_structs['individualPoints']. Si está presente, eso significa que hay un único lobo que transita por ese punto; de este lobo obtienen su individual-id y el node-id, además de sumar una unidad a la variable ‘trackingPoints’, que guarda la cantidad de puntos individuales de la ruta.	O(n)
Si el vértice no está en data_structs['individualPoints'], se busca en data_structs['MTPs'] y se suma una unidad a la cantidad de puntos de encuentro (‘totalMtps’).	O(n)
Se toma el valor de ‘entry’, una lista de diccionarios con los lobos que han pasado por ese punto de encuentro, para recorrer cada evento y obtener de allí el identificador de los lobos que han pasado por esa locación. Estos valores se guardan en la lista ‘wolfsIds’. Si la lista tiene más de seis elementos, se retornan los primeros y últimos tres identificadores de lobos; sino se retorna la cantidad encontrada.	O(n^2)
A ‘lstInfo’, una lista que guarda la información de cada nodo en la ruta, se le añade la longitud, latitud, id del nodo, los identificadores de los lobos que pasan por ese punto, la cantidad de lobos que transitan por esa locación, el siguiente vértice en la ruta y la distancia del punto inicial al punto final.	O(n)
A la lista ‘lstReturn’ se le añade la lista ‘lstInfo’ (que es una lista de listas con la información respectiva a cada nodo)	O(n)
El nodo de destino pasa a ser el nuevo nodo de inicio para cuando se lea el siguiente vértice de la ruta	O(n)
Se compara el valor de ‘lstReturn’ para retornar sólo los primeros cinco y últimos elementos (o aquellos disponibles)	
Se retorna el total de nodos del camino (se le quita una unidad por el último nodo agregado extra), los puntos de seguimiento, la distancia total, el total de puntos de	O(1)

encuentro y la lista a tabular con la información de los primeros y últimos 5 nodos del camino	
TOTAL	***O(n^2)***

Pruebas Realizadas

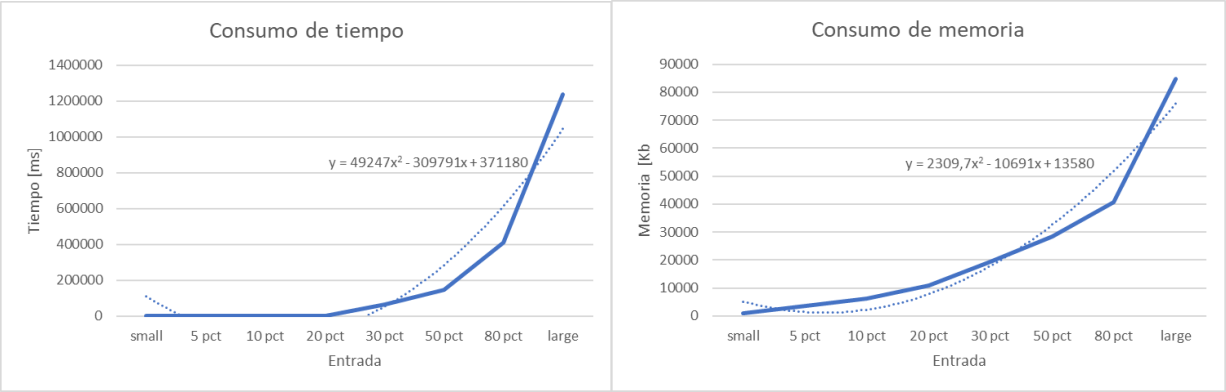
Descripción de las pruebas de tiempos de ejecución y memoria utilizada. Incluir descripción del procedimiento, las condiciones, las herramientas y recursos utilizados (librerías, computadores donde se ejecutan las pruebas, entre otros).

Los datos de entrada fueron 'm111p862_57p449','m111p908_57p427 '

Entrada	Memoria (kb)	Tiempo (s)
small	878.9013671875 kb	43.75499999523163 ms
5 pct	3505.4365234375 kb	162.01600003242493 ms
10 pct	6370.787109375 kb	687.2426999807358 ms
20 pct	10906.6826171875 kb	2290.2313998937607 ms
30 pct	19258.6064453125 kb	65217.73389995098 ms
50 pct	28542.724609375 kb	148159.4007999897 ms
80 pct	40741.951171875 kb	411735.50380015373 ms
large	84741.90234375kb	1235053.252670008 ms

Gráficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

En cuanto a la complejidad temporal, los resultados muestran una tendencia semejante a una gráfica de cuadrática, lo que coincide con los valores esperados. Lo anterior se debe en parte por el cálculo de la posible ruta entre un nodo inicial y un nodo final, pero sobre todo, por las iteraciones que se deben hacer por cada vértice perteneciente a la ruta identificada, ya que a partir de estos es que se itera sobre la lista que tienen asociada (que contiene los eventos ocurridos en esa locación) para obtener la información solicitada; es decir, la distancia total de la ruta, el total de puntos de encuentro y seguimiento, y los detalles de los primeros y últimos cinco vértices pertenecientes al camino. Respecto al uso de memoria, también puede observarse un comportamiento cuadrático, probablemente porque,

entre más registros disponibles se encuentren, mayor será la cantidad de nodos que se tengan que recorrer haciendo dfs, y mayor probabilidad hay de que el camino adquiera más puntos de recorrido.

Requerimiento 2

```
def req_2(data_structs, initialStation, destination):
    """
    Función que soluciona el requerimiento 2
    """
    # TODO: Realizar el requerimiento 2
    data_structs['search']= bfs.BreadhtFisrtSearch(data_structs['grafoDir'],initialStation)
    hay_camino= bfs.hasPathTo(data_structs['search'],destination)
    num_gathering=0
    tot_dist=0
    ult_nodo= None
    if hay_camino==True:
        lista_camino= lt.newList('ARRAY_LIST')
        lista_camino2= lt.newList('ARRAY_LIST')
        camino= bfs.pathTo(data_structs['search'],destination)
        i=1
        if camino is not None:
            num_vert = st.size(camino)
            while (not st.isEmpty(camino)):
                stop = st.pop(camino)
                lt.addLast(lista_camino, stop)
            for cada_nodo in lt.iterator(lista_camino):
                i+=1
                lista_stop= []
                node_id= cada_nodo
                if ult_nodo!= None and cada_nodo!= destination:
                    tot_dist+= gr.getEdge(data_structs['grafoDir'],cada_nodo, lt.getElement(lista_camino, i))['weight']
                    dist= gr.getEdge(data_structs['grafoDir'],cada_nodo,lt.getElement(lista_camino, i))['weight']
                else:
                    dist= 0.0
```

```
cant_= cada_nodo.count('_')
if cant_==1:
    num_gathering+=1
    individual_count0=gr.adjacentEdges(data_structs['grafoDir'],cada_nodo)
    individual_count= lt.size(individual_count0)
    info_stop = mp.get(data_structs['MTPs'],cada_nodo)
    info= me.getValue(info_stop)['elements'][0]
    long= info['location-long']
    lat=info['location-lat']
    lista_ady=[]
    for ady1 in lt.iterator(individual_count0):
        ad= ady1['vertexB']
        info_stop=mp.get(data_structs['individualPoints'], ad)
        info= me.getValue(info_stop)
        ady2= info['elements'][0]['individual-id']
        lista_ady.append(ady2)
    if len(lista_ady) > 6:
        lista_ady = ult3_prim3(lista_ady)
    else:
        lista_ady = lista_ady
    ady= lista_ady
else:
        info_stop=mp.get(data_structs['individualPoints'], cada_nodo)
        info= me.getValue(info_stop)
        individual_count= 1
        long= info['elements'][0]['location-long']
        lat=info['elements'][0]['location-lat']
        ady= info['elements'][0]['individual-id']
if cada_nodo!=destination:
```

```
                edge_to= lt.getElement(lista_camino, i)
            else:
                dist= 'Unknown'
                edge_to='Unknown'
            ult_nodo= cada_nodo
            lista_stop.append(long)
            lista_stop.append(lat)
            lista_stop.append(node_id)
            lista_stop.append(ady)
            lista_stop.append(individual_count)
            lista_stop.append(edge_to)
            lista_stop.append(dist)
            lt.addLast(lista_camino2,lista_stop)
    num_track= num_vert - num_gathering
    edges= num_vert- 1
    tot_dist= round(tot_dist, 4)
    if lt.size(lista_camino2) > 10:
        lista_camino2 = getiFirstandLast(lista_camino2,5)
    else:
        lista_camino2 = lista_camino2
    return lista_camino2, num_gathering, num_vert, num_track, edges, tot_dist
```

Descripción

Este requerimiento recibe como entrada el datastructs, el punto de inicio y el punto de destino, por medio del algoritmo bfs se encuentra la ruta con menos paradas, debido a que con este se busca la ruta con menos vértices vestidos, por medio de esto se encuentra la distancia recorrida, la cantidad de vértices, arcos y los puntos de encuentro.

Entrada	data_structs, initialStation, destination
Salidas	Distancia total del camino, total de puntos de encuentro en el camino, total de nodos de seguimiento en el camino, cinco primeros y cinco últimos vértices del camino con su respectiva información.
Implementado (Sí/No)	Si, por Angelica Ortiz

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	Complejidad
Se define el índice data_structs[‘search’] como una estructura para reconocer los vértices conectados al punto de partida	O(1)
A data_structs[‘search’] se le hace un recorrido dfs para saber si hay o no una ruta que conecte al punto de inicio con el punto de destino, y cuál es esta ruta.	O(V + E)

Se inicializa la variable de tot_dist, num_gathering y ult_nodo, que son respectivamente la distancia total el total de puntos de encuentro y la variable que guarda el ultimo nodo recorrido.	O(1)
Se crea la lista_camino y lista_camino2	O(1)
Si hay camino, se calcula la ruta a seguir desde el punto de inicio al punto final. Cada vértice que conforma el recorrido se añade a la lista_camino.	O(N)
Se empieza a recorrer la lista camino, a partir de este se toma la distancia y se toma el valor de este vértice en el mapa individual points y a partir de esto se toma el individual, id, latitud y longitud.	O(N)
Si es un vértice de un punto de encuentro se toma sus vértices adyacentes y a partir se recorren los vértices y de esto forma se toma el individual id, longitud y latitud	O(N^2)
Después se toma el vértice al cual va el recorrido	O(1)
Se añade cada uno de los valores anteriormente asignados a una lista del vértice para guardar esta información y se añade a la lista_camino2	O(8)
Finalmente se toma el número de vértices, de vértices de seguimiento, de arcos, de vértices de puntos de encuentro.	O(3)
TOTAL	***O(N^2)***

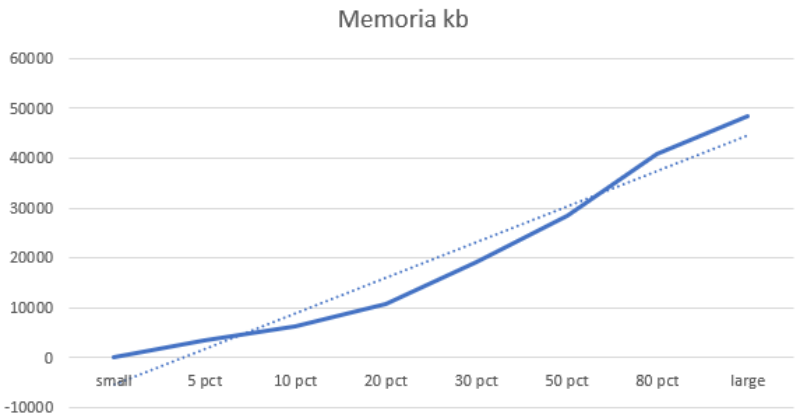
Pruebas Realizadas

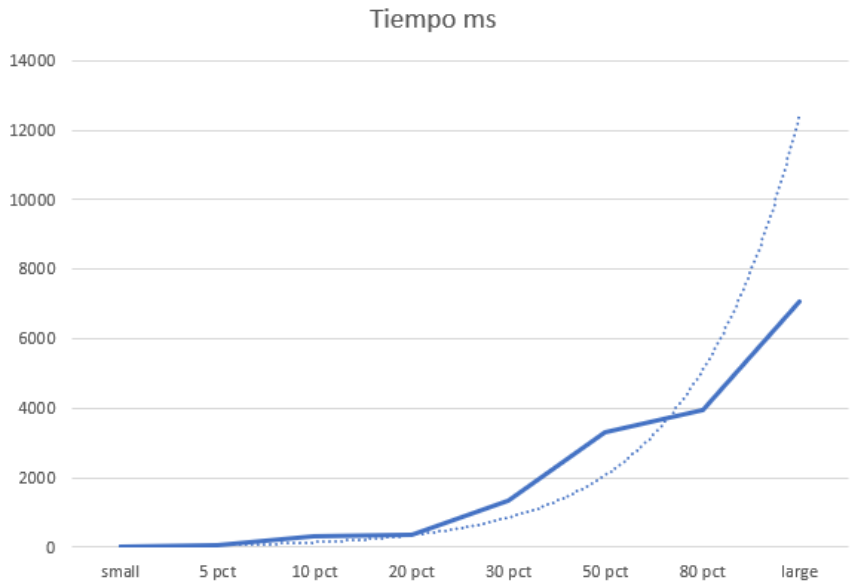
Descripción de las pruebas de tiempos de ejecución y memoria utilizada. Incluir descripción del procedimiento, las condiciones, las herramientas y recursos utilizados (librerías, computadores donde se ejecutan las pruebas, entre otros).

Entrada	Memoria	Tiempo (ms)
small	29.1728515625kb	10.913600087165833ms
5 pct	3507.5361328125kb	55.16040003299713ms
10 pct	6350.0078125kb	312.55770003795624ms
20 pct	10886.8671875kb	336.87690007686615ms
30 pct	19238.4375kb	1338.1383999586105ms
50 pct	28520.703125kb	3296.0746998786926ms
80 pct	40720.5078125kb	3934.1205999851227ms
large	48337.9423828125kb	7052.995100021362ms

Graficas

Las gráficas con la representación de las pruebas realizadas.





Análisis

Se pudo observar un crecimiento lineal con respecto a la memoria esto se pudo deber al recorrido realizado para encontrar el camino. Por otro lado, se pudo evidenciar un crecimiento semejante al cuadrático mostrando el reflejo de lo expuesto en el análisis de complejidad el cual era n^2 debido a que se debe recorrer dos veces los datos para encontrar los adyacentes.

Requerimiento 3

Objetivo: conocer los territorios de las manadas de lobos presentes dentro del hábitat del bosque.

Descripción

```python
def req_3(data_structs):
    """
    Función que soluciona el requerimiento 3
    """
    # TODO: Realizar el requerimiento 3
    sccStruct = scc.KosarajuSCC(data_structs['grafoDir'])
    numScc = scc.connectedComponents(sccStruct)
    idSccMap = mp.newMap(maptype='PROBING')
    for nodo in lt.iterator(mp.keySet(sccStruct['idscc'])):
        idScc = mp.get(sccStruct['idscc'],nodo)['value']
        contains = mp.get(idSccMap,idScc)
        if contains == None:
            lstNodes = lt.newList('ARRAY_LIST')
            lt.addLast(lstNodes,nodo)
            mp.put(idSccMap,idScc,lstNodes)
        else:
            lstNodes = me.getValue(contains)
            lt.addLast(lstNodes,nodo)

    omSize = om.newMap()
    for idScc in lt.iterator(mp.keySet(idSccMap)):
        lstNodes = me.getValue(mp.get(idSccMap,idScc))
        if lt.size(lstNodes) > 2:
            info = mp.newMap(maptype='PROBING')
            mp.put(info,'idScc',idScc)
            mp.put(info,'nodes',lstNodes)
            om.put(omSize,lt.size(lstNodes),info)
```

```python
c =0
FivemaxManInfo = []
while c < 5:
    minLat = 1000
    maxLat = 0
    minLong = 0
    maxLong = -1000
    infoScc = []
    maxM= om.maxKey(omSize)
    info = me.getValue(om.get(omSize,maxM))
    sccId = me.getValue(mp.get(info,'idScc'))
    lstNodesId = me.getValue(mp.get(info,'nodes'))
    nodeIds = getIFirstandLast(lstNodesId,3)
    wolfs = lt.newList('ARRAY_LIST')
    for nodo in lt.iterator(lstNodesId):
        entry = mp.get(data_structs['MTPs'],nodo)
        if entry == None:
            entry = mp.get(data_structs['individualPoints'],nodo)
            wolf = me.getValue(entry)['elements'][0]['individual-id']
            if not lt.isPresent(wolfs,wolf):
                lt.addLast(wolfs,wolf)
        event = me.getValue(entry)['elements'][0]
        if event['location-lat'] > maxLat:
            maxLat = event['location-lat']
        elif event['location-lat'] < minLat:
            minLat = event['location-lat']
        if event['location-long'] > maxLong:
            maxLong = event['location-long']
        elif event['location-long'] < minLong:
            minLong = event['location-long']
```

```python
infoScc.append(sccId)
res = []
p =0
for Id in nodeIds:
    lst = [Id]
    res.append(lst)
    p+= 1
    if p ==3:
        pt = ['...']
        res.append(pt)
infoScc.append(tabulate(res,tablefmt="plain"))
infoScc.append(maxM)
infoScc.append(minLat)
infoScc.append(maxLat)
infoScc.append(minLong)
infoScc.append(maxLong)
infoScc.append(lt.size(wolfs))
lstOflst =[]
if lt.size(wolfs) > 6:
    wolfs = getiFirstandLast(wolfs,3)
for lobo in lt.iterator(wolfs):
    infoLobo = []
    details = me.getValue(mp.get(data_structs['lobos'],lobo))
    infoLobo.append(details['individual-id'])
    infoLobo.append(details['animal-taxon'])
    infoLobo.append(details['animal-sex'])
    infoLobo.append(details['animal-life-stage'])
    infoLobo.append(details['study-site'])
    lstOflst.append(infoLobo)
```

Se calculan los cinco mayores componentes fuertemente conectados del grafo.

Entrada	No requiere parámetros de entrada
Salidas	Total de componentes fuertemente conectados y los cinco SCC mayores con su respectiva información
Implementado (Sí/No)	Sí. María José Amorocho

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	Complejidad
Se hace una estructura con los componentes conectados usando el algoritmo de Kosaraju sobre el grafo dirigido. La estructura se guarda en 'sccStruct'	O(n)
Se obtiene de la estructura el número de componentes fuertemente conectados.	O(1)
Se crea 'idSccMap', un mapa que guardará como llave el id del componente fuertemente conectado, y como valor una lista de vértices que pertenecen al id de ese componente.	O(1)
Para cada llave (vértice) en 'sccStruct['idscc']', se toma su valor (el Id del componente) y si no está en el mapa 'idSccMap', se agrega como una nueva llave. Si ya está en el mapa, se toma su valor: una lista una lista con los vértices (llaves) que pertenecen a ese Id, y se agrega la coordenada correspondiente.	O(n)
Se crea el mapa ordenado 'omSize' para guardar los cinco mayores componentes fuertemente conectados. Como llave tiene el tamaño del elemento fuertemente conectado analizado, y como valor tiene un mapa.	O(1)
Por cada Id (llave) que se encuentre en 'idSccMap' se extrae el tamaño de la lista que tiene asociada, que corresponden al número de vértices que tiene ese componente fuertemente conectado. Si la lista tiene más de dos elementos, se crea el mapa 'info', que tiene como llaves 'idScc', que guarda el id del componente fuertemente conectado, y 'nodes', una lista de vértices que hacen parte del componente. En el mapa ordenado 'omSize' se guarda como llave el tamaño de la lista con los vértices fuertemente conectados, y como valor, el mapa 'info'. De esta manera, 'omSize' queda organizado ascendentemente por los componentes fuertemente conectados con más elementos (los componentes fuertemente conectados con más vértices corresponden a las manadas con mayor dominio de territorio, pues justamente tienen más puntos de localización).	O(n)
Se inicializa un contador ('c') para obtener las cinco manadas con mayor dominio; es decir, los cinco primeros valores de 'omSize'.	O(1)
Para cada uno de los primeros cinco valores máximos de 'omSize' (cinco llaves mayores), se toma su valor, que es un mapa con el id del componente fuertemente conectado y la lista de posiciones que pertenecen al mismo. De allí se obtiene el número de puntos de encuentro y seguimiento que pertenecen a dicha manada (que corresponde al valor de la llave analizada) y las tres primeras y últimas localizaciones reconocidas en el componente.	O(5)
Para obtener los lobos que pertenecen a la manada, se busca cada vértice que pertenece al componente fuertemente conectado en data_structs['MTPs'] o data_structs['individualPoints']. Como estos mapas tienen como valor una lista de eventos por las que han pasado diferentes lobos, en extraer de cada evento el individual-id del lobo. Si el lobo no está presente en la lista 'wolfs', se añade, indicando que es un miembro de la manada.	O(n)
Del primer elemento en la lista de eventos pertenecientes a una localización, se obtiene la latitud y longitud, y se comparan con anteriores valores de longitud y latitud máxima y mínima para obtener justamente estos datos.	O(n)
Toda la información requerida por componente conectado (Id del componente conectado, los primeros tres y últimos vértices del componente, la longitud y latitud máxima y mínima, la cantidad de lobos que pertenecen a la manada y las especificaciones de los mismos) es añadida a la lista 'infoScc'. Al finalizar con los	O(5)

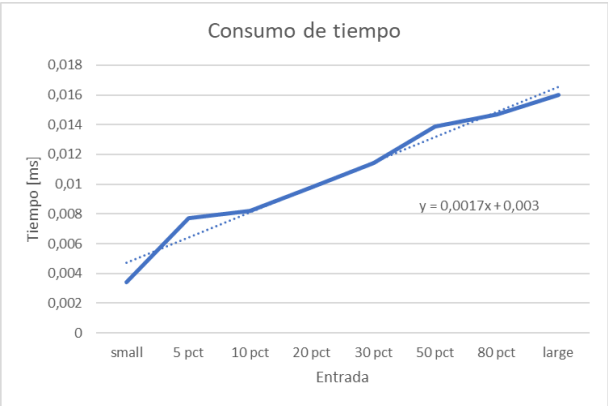
datos, ‘infoScc’ se añade a una lista de listas (‘FivemaxManInfo’). El contador ‘c’ se incrementa en una unidad y se elimina el elemento máximo de ‘omSize’ (para entonces procesar el siguiente elemento mayor).	
TOTAL	***O(n)***

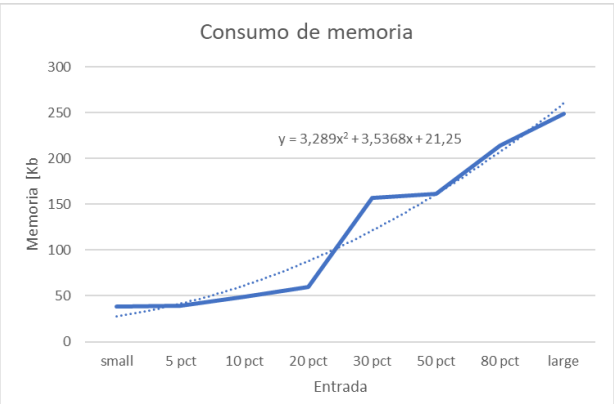
Pruebas Realizadas

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).

Entrada	Salida	Tiempo (s)	memoria
small	1389 componentes fuertemente conectados	0.003400087356567383 ms	38.4326171875kb
5 pct	1228 componentes fuertemente conectados	0.007700085639953613 ms	39.5048828125kb
10 pct	1019 componentes fuertemente conectados	0.008199981689453125 ms	48.685546875kb
20 pct	975 componentes fuertemente conectados	0.009799941062927246 ms	60.1171875kb
30 pct	933 componentes fuertemente conectados	0.011400061607360844 ms	156.775390625kb
50 pct	938 componentes fuertemente conectados	0.013900051116943359 4ms	161.318359375kb
80 pct	1053 componentes fuertemente conectados	0.014700017929077148 4ms	214.2685195695kb
large	1027 componentes fuertemente conectados	0.015999546051025391 ms	249.1746590831kb

Gráficas





Análisis

Respecto al consumo de memoria, se puede apreciar que se sigue una tendencia similar a O(n^2); se indaga que esto es dado a que, a mayor volumen de datos, se necesita calcular un mayor volumen

componentes fuertemente conectados, lo que implica que mayor va a ser el espacio usado en las estructuras de datos que ordenan la información por número de vértices y componentes fuertemente conectados. Por otro lado, el consumo de tiempo puede asemejarse a un comportamiento lineal, como se tenía previsto, pues, en el peor de los casos, los recorridos que se hacen son sobre la cantidad de llaves que contiene un mapa. Se cree que lo que ayuda a mejorar considerablemente el tiempo de ejecución es el uso de un mapa ordenado para obtener las manadas de mayor dominancia, pues con esta estructura, los datos se van organizando a medida que se introducen y su complejidad en el peor caso es de O(log n).

Requerimiento 4

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

```python
def req_4(data_structs, ini, fin):
    """
    Función que soluciona el requerimiento 4
    """
    # TODO: Realizar el requerimiento 4
    longini= float(ini[0])
    latini= float(ini[1])
    longfin= float(fin[0])
    latfin= float(fin[1])
    dist_menor_ini=9999999999
    mapa= data_structs['MTPs']
    keys= mp.keySet(mapa)
    ino=''
    dest=''
    for llave in lt.iterator(keys):
        lista_mtp= mp.get(mapa, llave)
        lista_mtp= me.getValue(lista_mtp)
        info= lt.getElement(lista_mtp, 1)
        long2=info['location-long']
        lat2= info['location-lat']
        dist= haversine(latini, longini, lat2, long2)
        if dist==0:
            continue
        if dist<dist_menor_ini:
            dist_menor_ini=dist
            ino=llave
    dist_menor_dest=999999999
    for llave in lt.iterator(keys):
            lista_mtp= mp.get(mapa, llave)
```

```
            lista_mtp= me.getValue(lista_mtp)
            info= lt.getElement(lista_mtp, 1)
            long2=info['location-long']
            lat2= info['location-lat']
            dist= haversine( lat2, long2, latfin, longfin)
            if dist==0:
                continue
            if dist<dist_menor_dest:
                dist_menor_dest=dist
                dest= llave
    info_stop = mp.get(data_structs['MTPs'],ino)
    info= me.getValue(info_stop)
    longino= info['elements'][0]['location-long']
    latino=info['elements'][0]['location-lat']
    individual_idino=gr.adjacentEdges(data_structs['grafoDir'],ino)
    lista_adyini= []
    for ady1 in lt.iterator(individual_idino):
        ad= ady1['vertexB']
        info_stop=mp.get(data_structs['individualPoints'], ad)
        info= me.getValue(info_stop)
        ady2= info['elements'][0]['individual-id']
        lista_adyini.append(ady2)
    individual_idino= lista_adyini

    info_stop = mp.get(data_structs['MTPs'],dest)
    info= me.getValue(info_stop)
    longdest= info['elements'][0]['location-long']
    latdest=info['elements'][0]['location-lat']
    individual_id_dest=gr.adjacentEdges(data_structs['grafoDir'],dest)
```

```
    lista_adyini= []
    for ady1 in lt.iterator(individual_id_dest):
        ad= ady1['vertexB']
        info_stop=mp.get(data_structs['individualPoints'], ad)
        info= me.getValue(info_stop)
        ady2= info['elements'][0]['individual-id']
        lista_adyini.append(ady2)
    individual_id_dest= lista_adyini
    paths= djk.Dijkstra(data_structs['grafoDir'], ino)
    hay_path= djk.hasPathTo(paths, dest)
    if hay_path ==True:
        path = djk.pathTo(paths, dest)
        lista_camino= lt.newList('ARRAY_LIST')
        lista_gathering= lt.newList('ARRAY_LIST')
        lista_camino2= lt.newList('ARRAY_LIST')
        dist_tot=0
        if path is not None:
            num_vert = st.size(path)
            while (not st.isEmpty(path)):
                stop = st.pop(path)
                lt.addLast(lista_camino, stop)
        for info in lt.iterator(lista_camino):
            lista= []
            src_node_id= info['vertexA']
            tgt_node_id= info['vertexB']
            dist= info[ 'weight']
            dist_tot+= info[ 'weight']
            src_count=src_node_id.count('_')
            tgt_count=tgt_node_id.count('_')
            if src_count==1:
```

```
        if src_count==1:
            lista_gath= []
            info_stop = mp.get(data_structs['MTPs'],src_node_id)
            info= me.getValue(info_stop)['elements'][0]
            longsrc= info['location-long']
            latsrc=info['location-lat']
            lt.addLast(lista_gathering, src_node_id)
        else:
            info_stop=mp.get(data_structs['individualPoints'], src_node_id)
            info= me.getValue(info_stop)
            longsrc= info['elements'][0]['location-long']
            latsrc=info['elements'][0]['location-lat']
        if tgt_count==1:
            info_stop = mp.get(data_structs['MTPs'],tgt_node_id)
            info= me.getValue(info_stop)['elements'][0]
            longtgt= info['location-long']
            lattgt=info['location-lat']
            individual_idtgt= info['individual-id']
            lt.addLast(lista_gathering, tgt_node_id)
        else:
            info_stop=mp.get(data_structs['individualPoints'], tgt_node_id)
            info= me.getValue(info_stop)
            longtgt= info['elements'][0]['location-long']
            lattgt=info['elements'][0]['location-lat']
            individual_idtgt= info['elements'][0]['individual-id']
        lista.append(src_node_id)
        lista.append(latsrc)
        lista.append(longsrc)
        lista.append(tgt_node_id)
```

```
                lista.append(tgt_node_id)
                lista.append(lattgt)
                lista.append(longtgt)
                lista.append(individual_idtgt)
                lista.append(dist)
                lt.addLast(lista_camino2, lista)
    listagathering= lt.newList('ARRAY_LIST')
    for mtp in lt.iterator(lista_gathering):
        lista1=[]
        info_stop = mp.get(data_structs['MTPs'],ino)
        info= me.getValue(info_stop)
        longmtp= info['elements'][0]['location-long']
        latmtp=info['elements'][0]['location-lat']
        individual_idmtp=gr.adjacentEdges(data_structs['grafoDir'],mtp)
        lista_adymtp= lt.newList('ARRAY_LIST')
        for ady1 in lt.iterator(individual_idmtp):
            ad= ady1['vertexB']
            info_stop=mp.get(data_structs['individualPoints'], ad)
            info= me.getValue(info_stop)
            ady2= info['elements'][0]['individual-id']
            lt.addLast(lista_adymtp,ady2)
        tam= lt.size(individual_idmtp)
        individual_idmtp= lista_adymtp
        if lt.size(individual_idmtp) > 6:
            individual_idmtp = getiFirstandLast(individual_idmtp,3)
        else:
            individual_idmtp = individual_idmtp
        lista1.append(mtp)
        lista1.append(longmtp)
```

```
    lista1.append(latmtp)
    lista1.append(individual_idmtp)
    lista1.append(tam)
    lt.addLast(listagathering, lista1)
num_arc= num_vert-1
size_gath= lt.size(lista_gathering)
tot_dist= dist_tot+ dist_menor_ini+ dist_menor_dest
if lt.size(listagathering) > 6:
    listagathering = getiFirstandLast(listagathering,3)
else:
    listagathering = listagathering
return lista_camino2, listagathering, num_arc, num_vert, size_gath, dist_tot, dist_menor_ini, dist_menor_dest, tot_dist, ino, longino, latino,indiv
```

Descripción

Este requerimiento recibe como entrada la localización geográfica del punto de origen Localización geográfica del punto de destino y el datastructs, a partir de esto encuentra los puntos de encuentro más cercanos a estas coordenadas ingresadas utilizando la función de haversine y al encontrarlos encuentra el camino con menor distancia entre estos por medio de estos encuentra la distancia.

Entrada	Localización geográfica del punto de origen (longitud y latitud). • Localización geográfica del punto de destino (longitud y latitud) y el datastructs

Salidas	La cantidad de arcos y vértices del camino, el recorrido realizado, la lista de los puntos de encuentro, la cantidad de puntos de encuentro, la distancia total, la distancia de las coordenadas ingresadas a los puntos de encuentro más cercanos, la longitud, latitud y node-id de los puntos de encuentro de las coordenadas ingresadas.
Implementado (Sí/No)	Si se implementó, por Angelica Ortiz

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Se toman las variables de latitud y longitud de las coordenadas ingresadas por el usuario	O(4)
Se inicializa la distancia menor inicial y distancia menor destino.	O(1)
Se toma el mapa de puntos de encuentro	O(1)
Se toman las llaves de mapa	O(1)
Se realiza el recorrido de las llaves y se toma el valor de la llave que se está recorriendo y se toma la longitud y latitud de esta y se realiza haversine y se compara para llegar a la distancia menor a tomar para tomar el vértice que presente esta distancia. Esto se realiza para el origen y el destino.	O(N)
Se toma el valor del vértice de origen y de este se toma la longitud latitud y sus adyacentes para tomar su individual id, este proceso se realiza con el vértice de destino también.	O(N)
Se define paths como una estructura para reconocer los vértices conectados al punto de partida con Dijkstra	O(1)
A paths se le hace un recorrido con Dijkstra para saber si hay o no una ruta que conecte al punto de inicio con el punto de destino, y cuál es esta ruta.	O(ElogV)
Se inicializan las variables de lista_camino, lista_camino2, distancia total y lista_gathering.	O(4)
Si hay camino, se calcula la ruta a seguir desde el punto de inicio al punto final. Cada vértice que conforma el recorrido se añade a la lista_camino.	O(N)
Se itera la lista camino y se toma la información de los vértices src y tgt y si es un punto de encuentro se crea una lista para este y se le añade la información y finalmente se agrega a la lista de gathering. Y se	O(N)

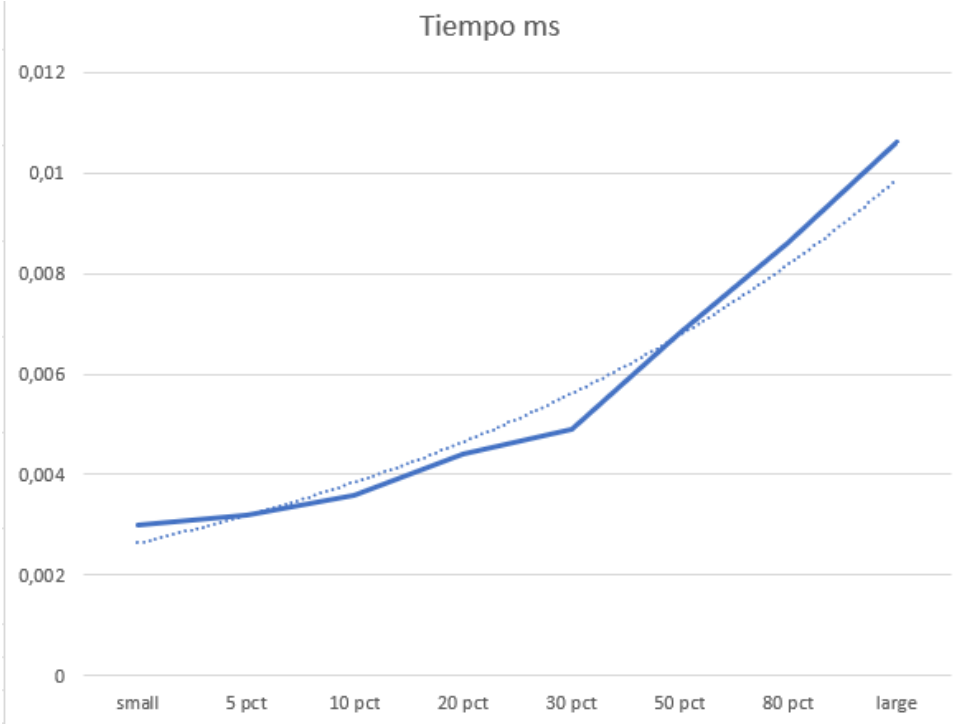
añade la información general a la lista del vértice y se añade a la de la lista camino2	
Se itera la lista de gathering para poder tener la información de cada uno de los puntos de encuentro y se itera por cada adyacente.	O(N^2)
Se toma el número de arcos, el número de puntos de encuentro y la distancia total	O(3)
TOTAL	***O(N^2)***

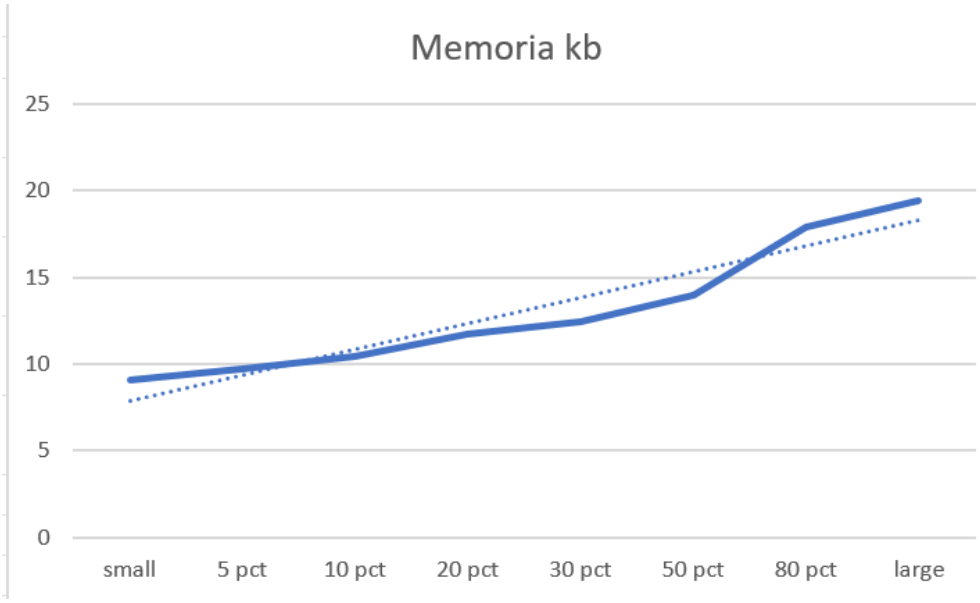
Pruebas Realizadas

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).

Entrada	Memoria(kb)	Tiempo (ms)
small	9.0546875kb	0.0030000020980834961ms
5 pct	9.7265625kb	0.0032000054168701172ms
10 pct	10.46875kb	0.0035999882125854492ms
20 pct	11.703125kb	0.0044000014877319336ms
30 pct	12.457628kb	0.0049000097846984863ms
50 pct	14.012845kb	0.0067999842834472656ms
80 pct	17.912742kb	0.0085999996566772461ms
large	19.419876kb	0,010626155675456357ms

Gráficas





Análisis

se puede observar con respecto a la memoria que esta presenta una tendencia lineal esto se puede deber a que está dependiendo de la cantidad de datos que son recorridos, que son parte del camino formado entre los dos vértices más cercanos a las coordenadas ingresadas por el usuario, por otro lado se puede observar un crecimiento cercano al cuadrático con respecto al tiempo ejecutado para realizar

la función esto se debe a que se realiza un for sobre un for sobre la misma cantidad de datos, generando una complejidad de o(n^2) como se pudo ver evidenciada en el análisis de complejidad.

Requerimiento 5

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

Descripción

La función “req_5” implementa el algoritmo de Dijkstra para encontrar los nodos alcanzables desde un punto de origen dentro de una distancia máxima predefinida. Luego, filtra los nodos según una cantidad mínima de puntos requeridos en el camino y devuelve una lista con información detallada sobre los caminos que cumplen con esta condición. El algoritmo de Dijkstra se utiliza para calcular las distancias mínimas entre nodos, generando una estructura de datos que almacena la información relevante. A continuación, se itera sobre los nodos alcanzables y se obtiene el camino hasta cada nodo utilizando la función “pathTo()”, construyendo una lista de puntos y contando su cantidad. Los nodos que cumplan con la cantidad mínima requerida de puntos se agregan a una lista final, que contiene detalles como la cantidad de puntos, la distancia total y la lista completa de puntos en el recorrido. En resumen, “req_5” combina el algoritmo de Dijkstra y un filtrado adicional para identificar y extraer los caminos que satisfacen las condiciones establecidas.

Entrada	El identificador del punto de encuentro de origen, la distancia máxima en kilómetros que puede recorrer el guardabosques, la mínima cantidad de puntos de encuentros que el guardabosques desea inspeccionar.
Salidas	El número máximo de rutas posibles para inspeccionar corredores migratorios. El corredor migratorio más extenso dentro del territorio.
Implementado (Sí/No)	Si se implementó, por Jacobo Zarruk.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

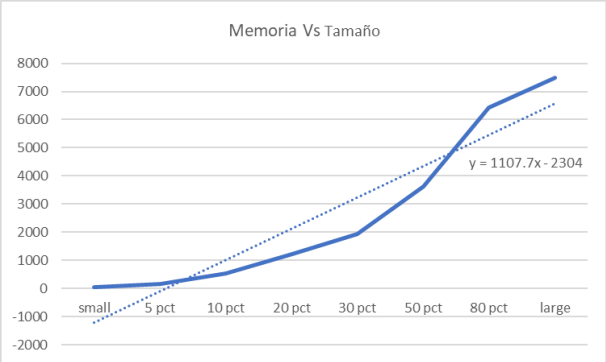
Pasos	**Complejidad**
Creación de la lista “listaNodos”	O(1)
Algoritmo de Dijkstra	O(E log (V))
Iteración sobre cada nodo alcanzable	O(N)
Agregar nodos alcanzables a “listaNodos”	O(1)
Creación de la lista “listaNodosCondiciones”	O(1)
Iteración sobre cada elemento en “listaNodos”	O(M)
Obtención del camino utilizando “pathTo()”	O(P)
Iteración sobre los elementos del camino	O(K)
TOTAL	***O(E log(V))***

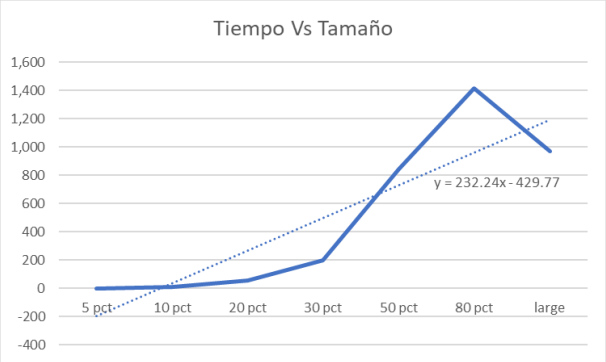
Pruebas Realizadas

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).

Entrada	Memoria (kb)	Tiempo (s)
small	64.48 kb	0,159 s
5 pct	175.09 kb	1,078 s
10 pct	526.87 kb	8,468 s
20 pct	1234.59 kb	57,146 s
30 pct	1931.03 kb	199,928 s
50 pct	3628.14 kb	843,269 s
80 pct	6414.07 kb	1416,858 s
large	7472.18 kb	967,722 s

Graficas





Análisis

El análisis de los datos muestra que tanto la memoria utilizada como el tiempo de ejecución aumentan significativamente a medida que se incrementa el tamaño de la entrada. Esto es esperado debido a la complejidad del algoritmo de Dijkstra utilizado en el código, que es de O(E log (V)), donde V es el número de vértices y E es el número de aristas en el grafo. El incremento no es lineal, sino logarítmico. La entrada "Large" muestra un salto considerable en la memoria y el tiempo, indicando un impacto significativo del tamaño del grafo en el rendimiento. En resumen, el aumento en la memoria y el tiempo es consistente con la complejidad del algoritmo y el crecimiento del grafo en función de los porcentajes de la entrada.

Requerimiento 6

Objetivo: identificar las diferencias de comportamiento de los lobos del estudio según el sexo registrado del individuo en determinado tiempo.

```python
def req_6(data_structs,animal_sex, ini, fin):
    """
    Función que soluciona el requerimiento 6
    """
    # TO DO: Realizar el requerimiento 6
    lista_lobos= lt.newList('ARRAY_LIST')
    lbs= data_structs['lobos']
    keys= mp.keySet(lbs)
    mapa_filt= mp.newMap(maptype='CHAINING')
    for lobo in lt.iterator(keys):
        entry= mp.get(lbs, lobo)
        info= me.getValue(entry)
        a_s= info['animal-sex']
        if a_s == animal_sex:
            lt.addLast(lista_lobos,lobo )
    mapa= data_structs['orderedData']
    for lobo in lt.iterator(lista_lobos):
        entry= mp.get(mapa, lobo)
        info= me.getValue(entry)
        for inf in lt.iterator(info):
            fecha= inf['timestamp']
            fecha= datetime.datetime.strptime(fecha,'%Y-%m-%d %H:%M')
            if fecha > ini and fecha< fin:
                if mp.contains(mapa_filt, lobo):
                    por_fecha=mp.get(mapa_filt, lobo)
                    por_fecha=me.getValue(por_fecha)
                    lt.addLast(por_fecha, inf)
                else:
                    por_anio= lt.newList('ARRAY_LIST')
                    lt.addLast(por_anio, inf)
                    mp.put(mapa_filt,lobo, por_anio)
```

```python
omDist = om.newMap()
for wolf in lt.iterator(mp.keySet(mapa_filt)):
    distance = 0
    lstEvents = me.getValue(mp.get(mapa_filt,wolf))
    lasttrack = None
    for event in lt.iterator(lstEvents):
        if lasttrack != None:
            startP = lasttrack['node-id']
            destP = event['node-id']
            distance += gr.getEdge(data_structs['grafoDir'],startP,destP)['weight']
        lasttrack = event
    om.put(omDist,distance,lstEvents)

maxDist = om.maxKey(omDist)
minDist = om.minKey(omDist)
wolfMaxDist = me.getValue(om.get(omDist,maxDist))['elements'][0]['individual-id']
wolfMinDist = me.getValue(om.get(omDist,minDist))['elements'][0]['individual-id']
maxWolfInfo = ObtainWolfInfo(data_structs,wolfMaxDist,maxDist)
minWolfInfo = ObtainWolfInfo(data_structs,wolfMinDist,minDist)
#segunda parte
pathInfoMax = pathInfo(data_structs,omDist,wolfMaxDist,maxDist)
pathInfoMin = pathInfo(data_structs,omDist,wolfMinDist,minDist)
```

Descripción

Breve descripción de como abordaron la implementación del requerimiento

Entrada	Fecha inicial, fecha final, sexo del animal
Salidas	El individuo que recorrió más distancia dentro del rango junto con su ruta más larga registrada, y el individuo que recorre menos distancia dentro del rango junto con su ruta registrada.
Implementado (Sí/No)	Sí. Implementado por Angélica y María José

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	Complejidad
Se define 'lista_lobos', una lista que guardará el id de los lobos que pertenezcan al sexo entrado por parámetro.	O(1)

Por cada llave (lobo) en data_structs['lobos'], se recorre el mapa comparando si el sexo del individuo es igual al sexo entrado por parámetro; de ser así, se añade el individual-id del lobo en 'lista_lobos'.	O(n)
Por cada lobo en 'lista_lobos', se busca su individual-id en el mapa data_structs['orderedData'], que tiene como valor una lista de posiciones recorridas por ese lobo. Por cada evento registrado en la lista, se compara el tiempo de ocurrencia con el rango dado por parámetro; si está dentro de las fechas indicadas, se guarda en 'mapa_filt', un mapa que tiene como llaves el individual-id de un lobo, y como valor, una lista con las posiciones filtradas por el rango de fechas.	O(n)
Se crea el mapa ordenado 'omDist', que tiene como llaves la distancia recorrida de cada lobo durante el tiempo establecido, y como valor, una lista con los trayectos de cada individuo en el rango de fechas dado.	O(1)
Por cada llave (lobo) en el mapa 'mapa_filt', se accede a la lista de posiciones que recorrió cada individuo dado el rango de tiempo. Se van tomando de a dos eventos para tomar su 'node-id', buscarlos en 'data_structs['grafoDir']' y obtener el arco que une a los dos vértices; del arco se toma su peso y se suma a la distancia recorrida del lobo. Al finalizar el proceso, se guarda en 'omDist' como llave, la distancia total obtenida, y como valor, la lista de eventos iterada.	O(n^2)
De 'omDist' se toman su llave máxima y mínima, que corresponden a la mayor distancia y menor distancia recorridas por algún lobo.	O(log n)
Tanto para el individuo con mayor y menor recorrido se obtiene su información específica con la función auxiliar 'ObtainWolfInfo', que retorna una lista con los detalles del animal.	O(1)
Con la función auxiliar 'pathInfo' se obtiene la información del camino que siguen el lobo con mayor recorrido, y el de menor recorrido. Esta función retorna el número de posiciones por las que pasaron, el total de arcos formados y una lista con la información de los primeros tres y últimos nodos de la ruta.	O(1)
De cada uno de los dos lobos, se retorna su individual-id, sus características individuales y la información del camino que siguen	O(1)
TOTAL	***O(n^2)***

Pruebas Realizadas

Las pruebas realizadas fueron realizadas en una máquina con las siguientes especificaciones. Los datos de entrada fueron el rango de fechas entre 2013-02-16 00:00 - 2014-10-23 23:59, y sexo femenino.

Procesadores	**Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz 2.10 GHz**
Memoria RAM	8 GB
Sistema Operativo	Windows 10

Entrada	Tiempo (ms)	Memoria (kb)
small	20.97	107.91

5 pct	66.47	108.84
10 pct	124.82	109.25
20 pct	215.32	109.26
30 pct	313.03	109.27
50 pct	562.17	112.78
80 pct	866.22	118.34
large	1015.92	126.58

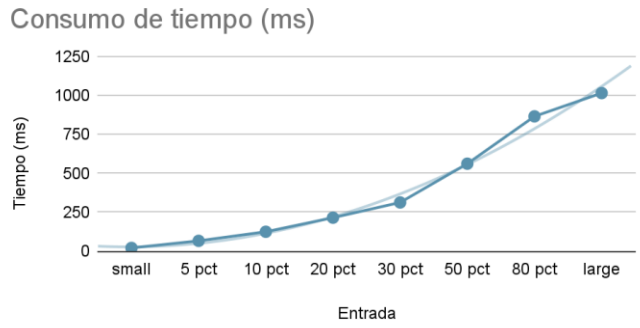
Tablas de datos

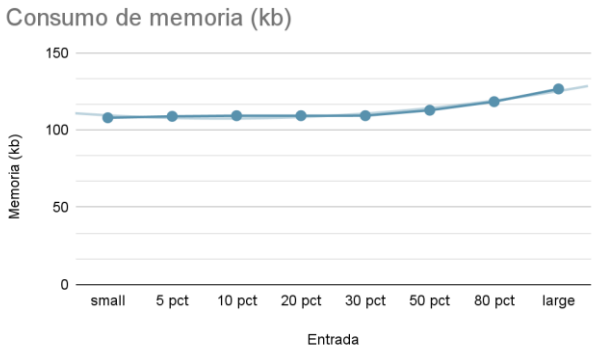
Las tablas con la recopilación de datos de las pruebas.

Muestra	Salida (Id de los lobos con mayor y menor recorrido)	Tiempo (ms)	Memoria (kb)
small	33680_33680 / 33669_33669	20.97	107.91
5 pct	33677_33677 / 32266_32266	66.47	108.84
10 pct	33677_33677 / 32266_32266	124.82	109.25
20 pct	33677_33677 / 32266_32266	215.32	109.26
30 pct	33677_33677 / 32266_32266	313.03	109.27
50 pct	33677_33677 / 32266_32266	562.17	112.78

80pct	33677_33677 / 32266_32266	866.22	118.34
large	33677_33677 / 32266_32266	1015.92	126.58

Gráficas





Análisis

Los resultados obtenidos coinciden con lo esperado teóricamente en cuanto al consumo de tiempo, pues se aprecia una tendencia de O(n^2). Lo anterior tiene coherencia con la forma en la que está estructurada el algoritmo, pues para filtrar los datos por sexo y rango de tiempo, por cada lobo presente se debe hacer una iteración sobre sus detalles personales, y así mismo, sobre cada una de as posiciones que tiene registradas para determinar si entran dentro del rango de tiempo solicitado, que es la parte del algoritmo con mayor complejidad. En cuanto al uso de memoria, se esperaba que este fuese mayor; indagamos que el uso tanto de un mapa ordenado como la filtración que se hace primeramente por sexo ayuda a que se optimice el uso de este elemento.

Requerimiento 7

```
def req_7(data_structs,dateI, dateF,tempMax,tempMin):
    """
    Función que soluciona el requerimiento 7
    """
    # TO DO: Realizar el requerimiento 7

    mapaFilt = mp.newMap()
    for wolf in lt.iterator(mp.keySet(data_structs['orderedData'])):
        lstWolfs = lt.newList('ARRAY_LIST')
        for track in lt.iterator(me.getValue(mp.get(data_structs['orderedData'],wolf))):
            fecha = track['timestamp']
            fecha= datetime.datetime.strptime(fecha,'%Y-%m-%d %H:%M')
            if float(track['external-temperature']) >= tempMin and float(track['external-temperature']) <= tempMax and fecha >= dateI and fecha <= da
                lt.addLast(lstWolfs,track)
        mp.put(mapaFilt,wolf,lstWolfs)

    mapaPosiciones = mp.newMap()
    graFilt = grafoFilt(mapaFilt,mapaPosiciones)
    for pos in lt.iterator(mp.keySet(mapaPosiciones)):
        if lt.size(me.getValue(mp.get(mapaPosiciones,pos))) >= 2:
            gr.insertVertex(graFilt,pos)
            for node in lt.iterator(me.getValue(mp.get(mapaPosiciones,pos))):
                gr.addEdge(graFilt,node['node-id'],pos,0)
                gr.addEdge(graFilt,pos,node['node-id'],0)
    #para los componentes
    sccStruct = scc.KosarajuSCC(graFilt)
    numScc = scc.connectedComponents(sccStruct)
    idSccMap = mp.newMap(maptype='PROBING')
    for nodo in lt.iterator(mp.keySet(sccStruct['idscc'])):
        idScc = mp.get(sccStruct['idscc'],nodo)['value']
```

```
    for nodo in lt.iterator(mp.keySet(sccStruct['idscc'])):
        idScc = mp.get(sccStruct['idscc'],nodo)['value']
        contains = mp.get(idSccMap,idScc)
        if contains == None:
            lstNodes = lt.newList('ARRAY_LIST')
            lt.addLast(lstNodes,nodo)
            mp.put(idSccMap,idScc,lstNodes)
        else:
            lstNodes = me.getValue(contains)
            lt.addLast(lstNodes,nodo)

    omSize = om.newMap()
    for idScc in lt.iterator(mp.keySet(idSccMap)):
        lstNodes = me.getValue(mp.get(idSccMap,idScc))
        if lt.size(lstNodes) > 2:
            info = mp.newMap(maptype='PROBING')
            mp.put(info,'idScc',idScc)
            mp.put(info,'nodes',lstNodes)
            om.put(omSize,lt.size(lstNodes),info)

    c =0
    FivemaxManInfo = []
    while c < 3:
        minLat = 1000
        maxLat = 0
        minLong = 0
        maxLong = -1000
        infoScc = []
        maxM= om.maxKey(omSize)
```

```
    info = me.getValue(om.get(omSize,maxM))
    sccId = me.getValue(mp.get(info,'idScc'))
    lstNodesId = me.getValue(mp.get(info,'nodes'))
    nodeIds = getIFirstandLast(lstNodesId,3)
    wolfs = lt.newList('ARRAY_LIST')
    for nodo in lt.iterator(lstNodesId):
        entry = mp.get(data_structs['MTPs'],nodo)
        if entry == None:
            entry = mp.get(data_structs['individualPoints'],nodo)
            wolf = me.getValue(entry)['elements'][0]['individual-id']
            if not lt.isPresent(wolfs,wolf):
                lt.addLast(wolfs,wolf)
        event = me.getValue(entry)['elements'][0]
        if event['location-lat'] > maxLat:
            maxLat = event['location-lat']
        elif event['location-lat'] < minLat:
            minLat = event['location-lat']
        if event['location-long'] > maxLong:
            maxLong = event['location-long']
        elif event['location-long'] < minLong:
            minLong = event['location-long']
    infoScc.append(sccId)
    res = []
    p =0
    for Id in nodeIds:
        lst = [Id]
        res.append(lst)
        p+= 1
        if p ==3:
```

```
                pt = ['...']
                res.append(pt)
        infoScc.append(tabulate(res,tablefmt="plain"))
        infoScc.append(maxM)
        infoScc.append(minLat)
        infoScc.append(maxLat)
        infoScc.append(minLong)
        infoScc.append(maxLong)
        infoScc.append(lt.size(wolfs))
        lstOflst =[]
        if lt.size(wolfs) > 6:
            wolfs = getiFirstandLast(wolfs,3)
        for lobo in lt.iterator(wolfs):
            infoLobo = []
            details = me.getValue(mp.get(data_structs['lobos'],lobo))
            infoLobo.append(details['individual-id'])
            infoLobo.append(details['animal-taxon'])
            infoLobo.append(details['animal-sex'])
            infoLobo.append(details['animal-life-stage'])
            infoLobo.append(details['study-site'])
            lstOflst.append(infoLobo)
        wolftable = tabulate(lstOflst,headers=['indiv-id','wolf taxon','wolf sex','life-stage','study-site'],
        |   |   |   |   |   |tablefmt='grid',maxheadercolwidths=5,maxcolwidths=5,stralign="center")
        infoScc.append(wolftable)
        FivemaxManInfo.append(infoScc)
        om.deleteMax(omSize)
        c += 1
    return numScc, FivemaxManInfo
```

Descripción

Breve descripción de cómo abordaron la implementación del requerimiento

Entrada	datastructs, la fecha inicial, la fecha final, la temperatura máxima y la temperatura final
Salidas	Respuesta esperada del algoritmo.
Implementado (Sí/No)	Si se implementó y quien lo hizo.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Se crea un mapa filtrado	O(1)
Se recorre cada llave del mapa ordereddata que es una llave del datastructs, se crea una lista y por cada recorrido realizado, se toma el timestamp y la temperatura y si estas se encuentran en el rango se añaden a la lista y se lade al mapa	O(N^2)
Se crea un mapaposiciones	O(1)
Se le asigna a la variable graffilt la función grafofilt la cual recibe el mapa filtrado y el mapa posiciones vacio	O(1)
En la función grafo filt, se crea un nuevo grafo el cual es dirijido, despues se recorre cada llave del mapa filtrado y se recorren a partir de esto se empieza a crear el grafo dirigido.	o(n)
Al ya tener el grafo se realiza el algoritmo de kosaraju para poder encontrar los componentes conectados	
TOTAL	***O(...)***

Pruebas Realizadas

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

Entrada	**Tiempo (s)**

Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

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.