Universidad de los Andes
Facultad de Ingeniería

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

ANÁLISIS RETO 4

Pedro Pablo Sanín, 202221527, p.sanin@uniandes.edu.co
Juan Esteban Rojas Olave, 202211481, je.rojaso1@uniandes.edu.co
Laura Andrea Hurtado Acosta, 202213259, la.hurtadoa1@uniandes.edu.co

ANÁLISIS DEL RETO

Laura Andrea Hurtado Acosta, código: 202213259, la.hurtadoa1@uniandes.edu.co

Requerimiento 5

Descripción

```
271  def req_5(data_structs, punto_partida, distancia, num_puntos):
272      """
273      Función que soluciona el requerimiento 5
274      """
275      # TODO: Realizar el requerimiento 5
276      distancia= round(float(distancia)/2, 3)
277      routes=djk.Dijkstra(data_structs["wolf_tracks_graph"], punto_partida)
278      validRoutes=om.newMap(omaptype="BST")
279
280      points = gr.vertices(data_structs["wolf_tracks_graph"])
281
282      for mpt in lt.iterator(points):
283          path = djk.pathTo(routes,mpt)
284          if path:
285              coste = round(djk.distTo(routes, mpt), 3)
286              numpoints = st.size(path)
287              if coste <= distancia and numpoints >= int(num_puntos):
288                  om.put(validRoutes, coste, mpt)
289
290      posibilities = om.size(validRoutes)
291
292      extensivePath = lt. newList()
293      corredores = om.keySet(validRoutes)
294
295      for distance in lt.iterator(corredores):
296          corredorPoint = om.get(validRoutes, distance)["value"]
297          path= djk.pathTo(routes, corredorPoint)
298          infopath = newPointExtensivePath(data_structs, path, distance, corredorPoint)
299          lt.addLast(extensivePath, infopath)
300
301      extensivePath= merg.sort(extensivePath, cmp_path_distance)
302      return {"extensivePath": extensivePath, "posibilities": posibilities}
303
```

Entrada	Entra como parámetro: punto de encuentro de origen, distancia en km que puede recorrer el guardabosques desde el origen, El número mínimo de puntos de encuentros que el guardabosques desea inspeccionar.

Salidas	• El número máximo de posibles de rutas para inspeccionar corredores migratorios. • El corredor migratorio más extensos dentro del territorio. Para este recorrido se debe mostrar la siguiente información: o El número de puntos de encuentro y seguimiento visitados. o La distancia recorrida (en km o m). o La secuencia de los puntos de encuentro e inspección involucrados en el trayecto. o La secuencia del número posible de individuos visibles en el trayecto.
Implementado (Sí/No)	Si. Implementado por Laura Hurtado

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Línea 276: se reduce la distancia que recorre el guardabosques en 2 teniendo en cuenta que él va y vuelve y se aproxima a 3 valores decimales.	O (1)
Línea 277: Se aplica el algoritmo de Dijkstra para organizar las rutas desde el punto de partida ingresado como parámetro.	O ((|E| + |V|) log |V|).
Línea 278: se validan las rutas organizadas en un mapa BST (binary search tree (BST)	O (1)
Línea 280: Por medio de la variable points, obtenemos los vértices del grafo de las “pistas” de los lobos.	O(|V|)
Línea 282: realizamos un recorrido para validar las rutas que vamos a seleccionar para el guardabosques, teniendo en cuenta el costo de los caminos y el numero de puntos, si es un punto valido se agrega al mapa creado anteriormente.	O (|V| + log n).
Línea 290: Las posibilidades va a ser el tamaño de las rutas validadas en el recorrido	O(n)
Línea 292: creamos una lista con la variable extensivepath que nos va a dar los caminos más extensos	O (1)
Línea 295: Realizamos un recorrido con el fin de encontrar las rutas mas extensas, evaluando los valores de las rutas que se validaron con anterioridad y se van agregando al final de la lista.	O (n + log n + |V|).
Línea 301: organizamos con el algoritmo de merge sort para que nos retorne la ruta más extensa.	O (n log n)
TOTAL	***O ((|E| + |V|) log |V|) + O (n log n)***

Pruebas Realizadas

Las pruebas realizadas fueron realizadas en una maquina con las siguientes especificaciones.

Procesadores	**Intel(R) Core (TM) i7-6500U CPU @ 2.50GHz 2.59 GHz**
Memoria RAM	12,0 GB (11,9 GB utilizable)
Sistema Operativo	Windows 10 Sistema operativo de 64 bits, procesador x64

Análisis

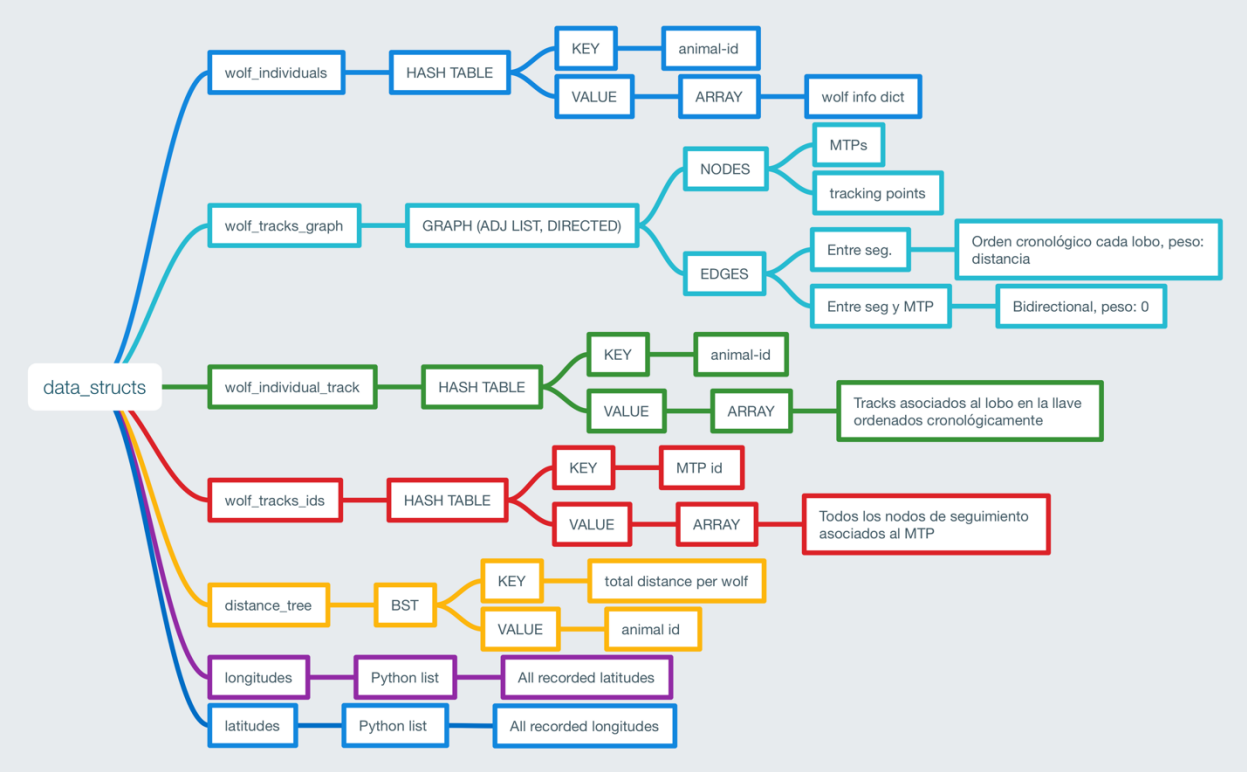
Al intentar realizar las pruebas correspondientes al requerimiento implementado hubo dos factores que afectaron el cumplimiento total del mismo:

Por un lado, la carga de datos e información de cada requerimiento en mi ordenador no fue posible por temas de recursión, llevando a que a la hora de correr cualquier requerimiento solo funcionara hasta el archivo de 10%, haciendo que los demás se saliera de la ejecución del programa, imposibilitando la continuación de las pruebas con la carga ya seleccionada.

Por otro lado, a pesar de tener algún error en el código que no deja visualizar la información que se espera se implementa usando el algoritmo de Dijkstra que posteriormente es organizado con un BST del cual parto para validar las rutas más extensas. Se puede observar en el análisis de complejidad que el término más dominante es O((|E| + |V|) log |V|), ya que es la única expresión que involucra tanto el número de aristas como el número de vértices del grafo, donde también tiene un mayor grado de complejidad en comparación con las otras expresiones, donde |E| es el número de aristas, |V| es el número de vértices y n es el número de elementos de la estructura de datos.

Aunque no se pudo realizar la grafica por falta de datos reales de la práctica se espera que el comportamiento de la gráfica sea exponencial, donde a mayor numero de datos, mayor sea el tiempo que requiere para ejecutar, teniendo en cuenta que N se va a hacer cada vez mayor, además de que los vértices y arcos van a aumentar en tamaños aumentando los recorridos de estos.

Carga de datos



(En controlador)

```
def load_data(mod_, filenames):
    """
    Carga los datos del reto
    """
    # TODO: Realizar la carga de datos

    mod = mod_['model']

    filename_individuals = filenames['individuals']
    file_individuals = csv.DictReader(open(filename_individuals, encoding = 'utf-8'),
delimiter = ',')

    filename_tracks = filenames['tracks']
    file_tracks = csv.DictReader(open(filename_tracks, encoding='utf-8'), delimiter =
',')

    latitudes = mod['latitudes']
    longitudes = mod['longitudes']

    start_time = get_time()
    rec_wolves = 0
    for wolf_individual in file_individuals:
        model.add_wolf_individual(mod, wolf_individual)
```

```
        rec_wolves += 1

    num_events = 0
    for wolf_track in file_tracks:
        num_events += 1
        model.add_wolf_track(mod, wolf_track)

    nums_vert1 = model.gr.numVertices(mod['wolf_tracks_graph'])
    nums_edg1 = model.gr.numEdges(mod['wolf_tracks_graph'])

    latitud_min = min(latitudes)
    latitud_max = max(latitudes)
    longitud_min = min(longitudes)
    longitud_max = max(longitudes)

    puntos_encuentro = mod['MTPs']




    model.trace_paths(mod)
    end_time = get_time()

    nums_edg2 = model.gr.numEdges(mod['wolf_tracks_graph'])

    list_first_last = model.first_last_n_elems_list(puntos_encuentro, 5)

    list_last_first_n_mtps = model.lt.newList('ARRAY_LIST')

    for elem in model.lt.iterator(list_first_last):
        dict_MTP = {}
        dict_MTP['Identificador'] = elem
        dict_MTP['Ubicación aprox'] = model.id_to_coords(elem)
        dict_MTP['Número de lobos en esa ubicación'] =
model.gr.outdegree(mod['wolf_tracks_graph'], elem)
        dict_MTP['Lobos adyacientes'] = []
        for elem in model.lt.iterator(model.gr.adjacents(mod['wolf_tracks_graph'],
elem)):
            dict_MTP['Lobos adyacientes'].append(elem)
        model.lt.addLast(list_last_first_n_mtps, dict_MTP)

    dict_result = {}
    dict_result['total_vertices'] = nums_vert1
    dict_result['total_aristas'] = nums_edg2
    dict_result['lobos_reconocidos'] = rec_wolves
    dict_result['eventos_cargados']= num_events
    dict_result['lobos_asociados_MTPs'] = int(nums_edg1/2)
    dict_result['num_puntos_encuentro'] = model.lt.size(puntos_encuentro)
```

```
    dict_result['arcos_seguimiento'] = nums_edg2-int(nums_edg1/2)
    dict_result['latitud_min'] = latitud_min
    dict_result['latitud_max'] = latitud_max
    dict_result['longitud_min'] = longitud_min
    dict_result['longitud_max'] = longitud_max
    dict_result['list_mtps'] = list_last_first_n_mtps
    dict_result['time'] = delta_time(start_time, end_time)


    return dict_result
```

(En modelo)

```
def new_data_structs():
    """
    Inicializa las estructuras de datos del modelo. Las crea de
    manera vacía para posteriormente almacenar la información.
    """
    #TODO: Inicializar las estructuras de datos
    data_structs = {}
    data_structs['wolf_individuals'] = mp.newMap(numelements=22, maptype='PROBING',
loadfactor=0.5)
    data_structs['wolf_tracks_graph'] = gr.newGraph(datastructure='ADJ_LIST', directed
= True, size= 200000)
    data_structs['wolf_individual_track'] = mp.newMap(numelements = 22,
maptype='PROBING', loadfactor=0.5)
    data_structs['wolf_tracks_ids'] = mp.newMap(numelements= 50000, maptype='PROBING',
loadfactor=0.5)
    data_structs['MTPs'] = lt.newList(datastructure='ARRAY_LIST',
cmpfunction=standard_compare)
    data_structs['all_tracks'] = lt.newList(datastructure='ARRAY_LIST',
cmpfunction=standard_compare)
    data_structs['distance_tree'] = om.newMap('BST', standard_compare)
    data_structs['longitudes'] = []
    data_structs['latitudes'] = []
    return data_structs
```

```
def add_wolf_individual(data_structs, wolf_data):
    """
    Función para agregar nuevos elementos a la lista
    """
    #TODO: Crear la función para agregar elementos a una lista
    individual_id = str(wolf_data['animal-id']) + '_' + str(wolf_data['tag-id'])
    wolf_data['individual-id'] = individual_id
    wolf_individuals = data_structs['wolf_individuals']
    if not mp.contains(wolf_individuals, individual_id):
        mp.put(wolf_individuals, individual_id, wolf_data)
```

```
def add_wolf_track(data_structs, wolf_track):
    timestamp_str = wolf_track['timestamp']
    wolf_track['timestamp'] = str_to_datetime(timestamp_str)

    wolf_track_lat_str = wolf_track['location-lat']
    wolf_track['location-lat'] = round_up_str(wolf_track_lat_str,3)
    wolf_track_long_str = wolf_track['location-long']
    wolf_track['location-long'] = round_up_str(wolf_track_long_str,3)

    data_structs['longitudes'].append(wolf_track['location-long'])
    data_structs['latitudes'].append(wolf_track['location-lat'])

    individual_id = str(wolf_track['individual-local-identifier']) + '_' +
str(wolf_track['tag-local-identifier'])
    wolf_track['individual-id'] = individual_id



    lt.addLast(data_structs['all_tracks'], wolf_track)

    add_seg(data_structs, wolf_track)
```

```
def add_seg (data_structs, wolf_track, is_directed = True):
    graph = data_structs['wolf_tracks_graph']
    individual_id = wolf_track['individual-id']
    wolf_track_MTP = id_MTP(wolf_track['location-long'], wolf_track['location-lat'])
    wolf_track_seg_id = id_seg(wolf_track['location-long'], wolf_track['location-
lat'], individual_id)
    wolf_track['animal-seg-id'] = wolf_track_seg_id

    add_to_hash_table(data_structs['wolf_tracks_ids'], wolf_track_MTP,
wolf_track_seg_id)
    add_to_hash_table(data_structs['wolf_individual_track'], individual_id,
wolf_track, compare_wolf_track_datetime)
    if not gr.containsVertex(graph, wolf_track_seg_id):

        insert_vertex(graph, wolf_track_seg_id)

        list_MTP = me.getValue(mp.get(data_structs['wolf_tracks_ids'],
wolf_track_MTP))
        num_dif_elems_MTP = lt.size(list_MTP)

        if num_dif_elems_MTP > 1:
            if not gr.containsVertex(graph, wolf_track_MTP):
                insert_vertex(graph, wolf_track_MTP)
                lt.addLast(data_structs['MTPs'], wolf_track_MTP)
                for elem in lt.iterator(lt.subList(list_MTP, 1, lt.size(list_MTP)-1)):
```

```
                    add_edge(graph, wolf_track_MTP, elem)
                    if is_directed:
                        add_edge(graph, elem, wolf_track_MTP)
            add_edge(graph, wolf_track_MTP, wolf_track_seg_id)
            if is_directed:
                add_edge(graph, wolf_track_seg_id, wolf_track_MTP)
```

```
def trace_paths(data_structs):
    graph = data_structs['wolf_tracks_graph']
    track_wolves = data_structs['wolf_individual_track']
    tree_distances = data_structs['distance_tree']
    for wolf_compound_id in lt.iterator(mp.keySet(track_wolves)):
        wolf_list = me.getValue(mp.get(track_wolves, wolf_compound_id))
        merg.sort(wolf_list, sort_criteria_datetime)
        total_distance = 0
        count = 0
        size = lt.size(wolf_list)
        for elem1 in lt.iterator(wolf_list):
            count += 1
            if count != size:
                elem2 = lt.getElement(wolf_list, count + 1)

                elem1_long = elem1['location-long']
                elem1_lat = elem1['location-lat']
                elem2_long = elem2['location-long']
                elem2_lat = elem2['location-lat']

                distance = haversine(elem1_long, elem1_lat, elem2_long, elem2_lat)

                if distance > 0 and (elem1['timestamp'] != elem2['timestamp']):
                    id_seg1 = elem1['animal-seg-id']
                    id_seg2 = elem2['animal-seg-id']
                    if gr.getEdge(graph, id_seg1, id_seg2) == None:
                        total_distance += distance
                    add_edge(graph, id_seg1, id_seg2, distance)
        tree_distances = om.put(tree_distances, total_distance, wolf_compound_id)
```

Descripción

En la carga de datos leemos dos archivos, el archivo con la información de los lobos y el archivo con la información con los tracks de seguimiento.

Entrada	El archivo con la información de los lobos y el archivo de de
Salidas	Una estructura de datos con: una tabla de hash con la información de los lobos (basada en el archivo de información de lobos), un grafo dirigido con la información de seguimiento de los lobos, una tabla de hash con los datos de seguimiento asociados a cada lobo ordenados cronológicamente, una tabla de hash con los MTPs (tanto hipotéticos como reales) y sus

	respectivos puntos de seguimiento, una lista con los MTPs verdaderos insertados en el grafo, una lista con todos los tracks de lobos, un árbol de búsqueda binaria para guardar y ordenar las distancias recorridas por los lobos y finalmente dos listas, una con latitudes y otra con longitudes para definir los rangos de posición de los lobos.
Implementado (Sí/No)	Sí

Análisis de complejidad

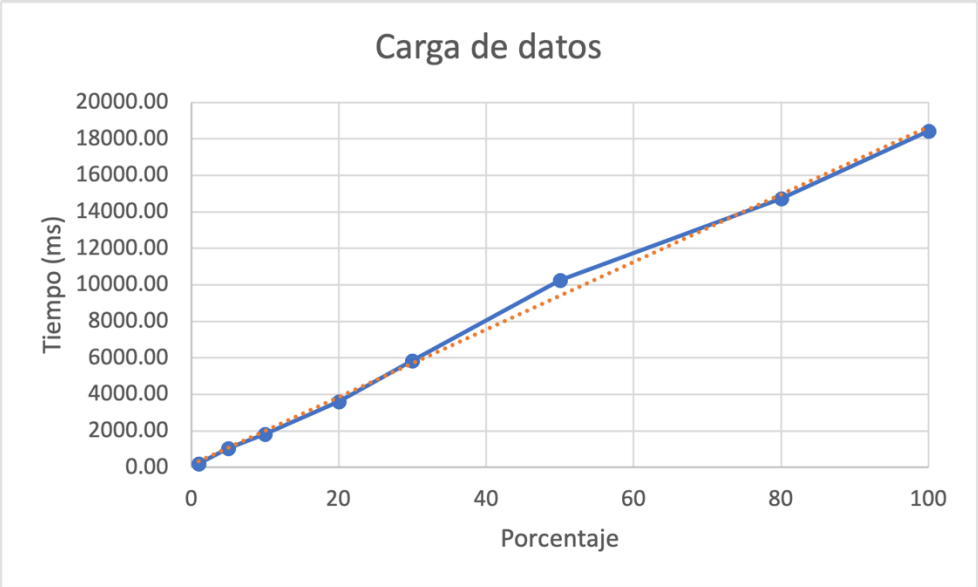
Pasos	**Complejidad**
P1: Se inicializa la función load_data en el controlador y se configuran los archivos de individuos y tracks para poder ser leídas por python utilizando la función csv.DictReader (controlador 51 -57)	O(1)
P2: Se itera sobre el archivo de información de individuos (file_individuals). En cada iteración se invoca la función del modelo add_wolf_individual. (64-66)	O(1) (Hay 45 lobos en todos los archivos)
P2.1: Se crea el id individual que identifica cada lobo a partir de el id del animal y el id del localizador. Se añade la información a la tabla de hash con la información de los lobos (567-576 model)	O(1)
P3: Se itera sobre el archivo de tracks y en cada iteración se invoca la función del modelo add_wolf_track	O(V)
P3.1: Se formatean los timestamps, se redondean las posiciones a 3 decimales y se crea el identificador individual para cada track a partir del id de animal y el id de localizador. Se añade el track a la lista con todos los tracks (datastructs[‘all_tracks’] (544-560)	O(1)
P3.2: Se invoca la función add_seg (562)	
P3.2.1: Se crea el vértice asociado al track. Se calcula el MTP hipotético para el track. Se añade el track a la tabla de hash con los tracks para cada lobo ([‘wolf_individual_track’]) y a la tabla de hash con los puntos de seguimiento para cada MTP hipotético. Las llaves se basan en el código individual del lobo y el código de MTP hipotético calculado anteriormente. (521-522)	O(1)
P3.2.2: Se revisa que el grafo no contenga el vértice, esto asegura que no hayan vértices del mismo lobo en una misma ubicación. Se añade el vértice al grafo ([‘wolf_tracks_graph’]), luego se saca la lista con todos los nodos en el mismo sitio para determinar si se debe insertar un MTP en esa ubicación. Si la lista con los puntos de seguimiento en esa ubicación tiene 2 entradas de lobos diferentes, se añade el MTP y se conecta con el primer elemento guardado en esa ubicación y el elemento guardándose. De lo contrario, si tiene más de dos elementos, conecta el vértice siendo guardado con su MTP correspondiente con una conexión bidireccional de peso 0. (523-540)	O(1)
P4: En el controlador, se revisan estadísticas del grafo antes de terminarlo de conectar.(73-81)	O(1)

P5: Se invoca la función del modelo trace paths para conectar el grafo.	O(V*log(V)+ E)
P5.1: Se recorre la tabla de hash con los registros de lobos asociados a cada individuo.	O(V*log(V))
P5.1.1: Se ordena el conjunto de tracks en orden cronológico utilizando el organismo mergesort. (490)	O(V*log(V))
P5.1.2: Se inicializan variables para registrar la distancia total recorrida por el lobo. (491-493)	O(1)
P5.1.3: Se itera sobre la lista ordenada de tracks	O(E)
P5.1.3.1: Se calcula la distancia con el siguiente track y se crea la arista correspondiente en el grafo, con peso igual a la distancia entre los dos puntos calculada usando la formula de haversine. Si la distancia es 0, se conectan esos dos puntos. Se actualiza la variable con la distancia total recorrida por el lobo. (488-511)	O(1)
P5.1.4: Se añade la distancia al árbol BST como llave, con valor igual al código de el individuo.(512)	O(1)
P6: En el controlador se obtiene el resto de información sobre el grafo.	O(1)
TOTAL	***O(V+V*log(V)+E)***

Pruebas

Procesador	1,4 GHz Intel Core i5 de cuatro núcleos
Sistema operativo	MacOS Ventura 13.2.1

Entrada (pct)	**Tiempo (ms)**
1	221.61
5	1039.08
10	1836.44
20	3599.24
30	5839.22
50	10276.22
80	14747.53
100	18451.85



Análisis de resultados

En la práctica podemos ver que nuestra carga de datos siguió realmente un orden de crecimiento lineal. Cabe resaltar que de ahora en adelante no diferenciaremos entre vértices y aristas al momento de hacer los análisis de complejidad, pues el número de vértices y aristas es casi igual para todas las muestras de datos. La diferencia entre los resultados experimentales y teóricos radica, seguramente en la complejidad de los ordenamientos que deben hacerse para cada lobo. Si bien al final se ordenan todos los vértices del grafo, no es lo mismo ordenar una sola lista co todos los vértices que ordenar varias listas pequeñas con subconjuntos de los mismos. El segundo procedimiento es mucho menos complejo y por ende los ordenamientos no generaron un crecimiento desmedido del orden de complejidad temporal.

Requerimiento 1

En este requerimiento se busca encontrar un camino entre dos vértices pasados por parámetro.

Entrada	La estructura de datos creada en la carga
Salidas	Una lista con los primeros y últimos 5 puntos del camino encontrado, la distancia total recorrida, el número de puntos de encuentro en el camino y el camino encontrado
Implementado (Sí/No)	Sí

```
def req_1(data_structs, punto_partida, punto_llegada):
    """
    Función que soluciona el requerimiento 1
    """
    # TODO: Realizar el requerimiento 1

    grafo = data_structs['wolf_tracks_graph']
    dfs_structs = dfs.DepthFirstSearch(grafo, punto_partida)
    path = dfs.pathTo(dfs_structs, punto_llegada)
    if path != None:
        list_result = lt.newList('ARRAY_LIST')
        distancia_total = 0
```

```
        puntos_encuentro = 0
        index = 1
        while index <= lt.size(path):
            elem = lt.getElement(path, index)
            dict_elem = {}
            dict_elem['Identificador'] = elem
            dict_elem['Latitud'] = id_to_coords(elem)[0]
            dict_elem['Longitud'] = id_to_coords(elem)[1]
            puntos_encuentro += 1
            if index != lt.size(path):
                next_elem = lt.getElement(path, index + 1)
                next_elem_lat = id_to_coords(next_elem)[0]
                next_elem_long = id_to_coords(next_elem)[1]
                distance =
haversine(float(dict_elem['Longitud']),float(dict_elem['Latitud']),
float(next_elem_long), float(next_elem_lat))
                dict_elem['Distancia al próximo lobo'] = distance
                distancia_total += distance
            else:
                dict_elem['Distancia al próximo lobo'] = '--'
            if is_MTP(elem):
                first_last_n_elems_list(gr.adjacents(grafo, elem), 3)
                wolves_MTP = gr.adjacents(grafo, elem)
                list_wolves_transit = first_last_n_elems_list(wolves_MTP, 3)
                list_wolves = []
                for wolf in lt.iterator(list_wolves_transit):
                    list_wolves.append(wolf)
                dict_elem['Primeros y últimos tres lobos que transitan por el punto']
= list_wolves
            else:
                dict_elem['Primeros y últimos tres lobos que transitan por el punto']
= 'Desconocido'
            lt.addLast(list_result, dict_elem)
            index += 1
        return first_last_n_elems_list(list_result, 5), distancia_total,
puntos_encuentro, path
    return lt.newList(), 0, 0

def req_2(data_structs, punto_partida, punto_llegada):
    """
    Función que soluciona el requerimiento 2
    """
    # TODO: Realizar el requerimiento 2
    grafo = data_structs['wolf_tracks_graph']
    bfs_structs = bfs.BreadthFirstSearch(grafo, punto_partida)
    path = bfs.pathTo(bfs_structs, punto_llegada)
    if path != None:
        list_result = lt.newList('ARRAY_LIST')
```

```
        distancia_total = 0
        puntos_encuentro = 0
        index = 1
        while index <= lt.size(path):
            elem = lt.getElement(path, index)
            dict_elem = {}
            dict_elem['Identificador'] = elem
            dict_elem['Latitud'] = id_to_coords(elem)[0]
            dict_elem['Longitud'] = id_to_coords(elem)[1]
            puntos_encuentro += 1
            if index != lt.size(path):
                next_elem = lt.getElement(path, index + 1)
                next_elem_lat = id_to_coords(next_elem)[0]
                next_elem_long = id_to_coords(next_elem)[1]
                distance =
haversine(float(dict_elem['Longitud']),float(dict_elem['Latitud']),
float(next_elem_long), float(next_elem_lat))
                dict_elem['Distancia al próximo lobo'] = distance
                distancia_total += distance
            if is_MTP(elem):
                first_last_n_elems_list(gr.adjacents(grafo, elem), 3)
                wolves_MTP = gr.adjacents(grafo, elem)
                list_wolves_transit = first_last_n_elems_list(wolves_MTP, 3)
                list_wolves = []
                for wolf in lt.iterator(list_wolves_transit):
                    list_wolves.append(wolf)
                dict_elem['Primeros y últimos tres lobos que transitan por el punto']
= list_wolves
            else:
                dict_elem['Primeros y últimos tres lobos que transitan por el punto']
= 'Desconocido'
            lt.addLast(list_result, dict_elem)
            index += 1
        return first_last_n_elems_list(list_result, 5), distancia_total,
puntos_encuentro, path
    return lt.newList(), 0, 0
```

Análisis de complejidad

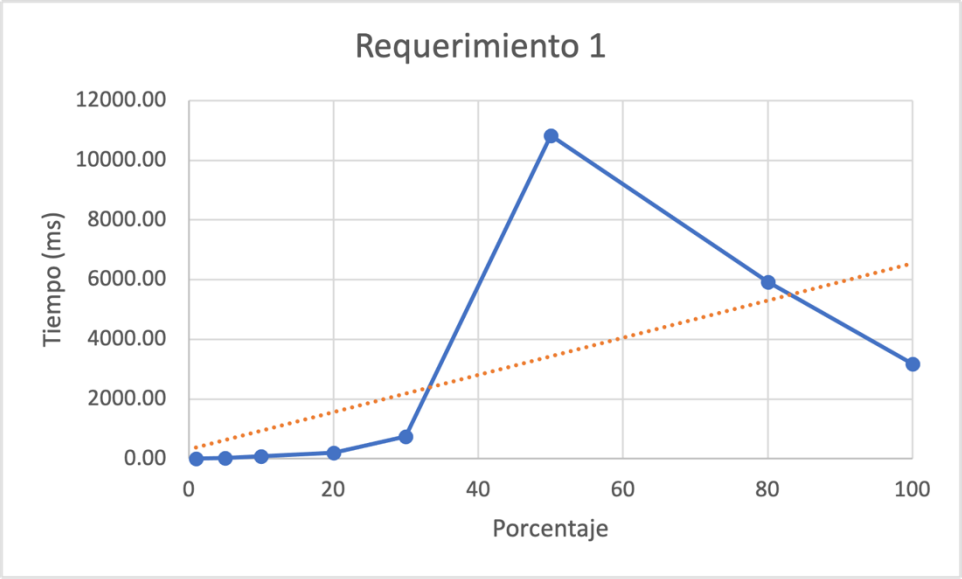
Pasos	Complejidad
P1: Correr el algoritmo DFS a partir del punto de partida. Se crea la estructura dfs_structs (92)	**O(V+E)**
P2: Con dicha estructura utilizamos la función dfs.pathTo para encontrar un camino entre los dos puntos.	**O(1)**
P3: Si existe un camino comenzamos a encontrar su información para presentarla al final del requerimiento.	**--**

P3.1: Creamos una lista para guardar el resultado (list_result) y creamos variables para guardar la distancia total recorrida y el número de puntos de encuentro (MTPs) en el camino. (95-98)	**O(1)**
P3.2: Por cada elemento del camino creamos un diccionario en el cual guardamos sus coordenadas, e identificador. También revisamos el próximo lobo y calculamos sus distancias, guardamos esta información en el diccionario y actualizamos la variable de distancia total.	**O(1)**
P3.3: Revisamos si el punto es un MTP, si lo es, miramos sus adyacentes y guardamos la informacion de los primeros y últimos lobos que pasaron por ese punto, de lo contrario, guardamos 'Desconocido'.	**O(1)**
P3.4: Guardamos el diccionario en list_result, tomamos sus primeros y últimos tres elementos y retornamos esa lista, el número de puntos de encuentro, la distancia total recorrida el path completo (será de utilidad al momento de crear el mapa)	**O(1)**
TOTAL	***O(V+E)***

Pruebas

Procesador	1,4 GHz Intel Core i5 de cuatro núcleos
Sistema operativo	MacOS Ventura 13.2.1
Punto inicial	m111p862_57p449
Punto final	m111p908_57p427

Entrada (pct)	**Tiempo (ms)**
1	5.26
5	29.76
10	92.56
20	202.41
30	747.25
50	10833.94
80	5930.76
100	3171.10



Análisis de complejidad

Este resultado, si bien puede parecer errado, tiene muchísimo sentido. En DFS, el camino que se toma entre dos vértices es prácticamente aleatorio, de ahí que la iteración sobre los vértices del camino puede tener complejidades radicalmente distintas y completamente independientes al tamaño de los datos. En BFS, este problema no debería ocurrir o al menos debería ser menos pronunciado pues en ese algoritmo, el camino es el más corto entre los dos nodos.

Requerimiento 2

En este requerimiento se busca encontrar el camino más cort entre dos vértices pasados por parámetro.

Entrada	La estructura de datos creada en la carga, el punto inicial y el punto final
Salidas	Una lista con los primeros y últimos 5 puntos del camino encontrado, la distancia total recorrida, el número de puntos de encuentro en el camino y el camino encontrado
Implementado (Sí/No)	Sí

```
def req_2(data_structs, punto_partida, punto_llegada):
    """
    Función que soluciona el requerimiento 2
    """
    # TODO: Realizar el requerimiento 2
    grafo = data_structs['wolf_tracks_graph']
    bfs_structs = bfs.BreadthFirstSearch(grafo, punto_partida)
    path = bfs.pathTo(bfs_structs, punto_llegada)
    if path != None:
        list_result = lt.newList('ARRAY_LIST')
        distancia_total = 0
        puntos_encuentro = 0
        index = 1
        while index <= lt.size(path):
            elem = lt.getElement(path, index)
            dict_elem = {}
            dict_elem['Identificador'] = elem
            dict_elem['Latitud'] = id_to_coords(elem)[0]
            dict_elem['Longitud'] = id_to_coords(elem)[1]
            puntos_encuentro += 1
            if index != lt.size(path):
                next_elem = lt.getElement(path, index + 1)
                next_elem_lat = id_to_coords(next_elem)[0]
                next_elem_long = id_to_coords(next_elem)[1]
                distance =
haversine(float(dict_elem['Longitud']),float(dict_elem['Latitud']),
float(next_elem_long), float(next_elem_lat))
                dict_elem['Distancia al próximo lobo'] = distance
                distancia_total += distance
            if is_MTP(elem):
                first_last_n_elems_list(gr.adjacents(grafo, elem), 3)
                wolves_MTP = gr.adjacents(grafo, elem)
                list_wolves_transit = first_last_n_elems_list(wolves_MTP, 3)
                list_wolves = []
                for wolf in lt.iterator(list_wolves_transit):
                    list_wolves.append(wolf)
```

```
                dict_elem['Primeros y últimos tres lobos que transitan por el punto']
= list_wolves
            else:
                dict_elem['Primeros y últimos tres lobos que transitan por el punto']
= 'Desconocido'
            lt.addLast(list_result, dict_elem)
            index += 1
        return first_last_n_elems_list(list_result, 5), distancia_total,
puntos_encuentro, path
    return lt.newList(), 0, 0
```

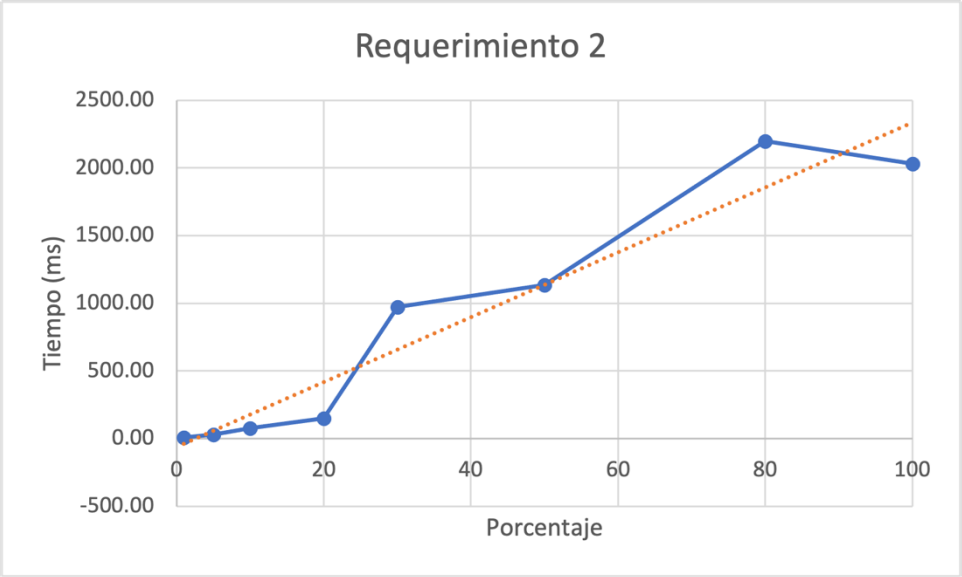
Pasos	**Complejidad**
P1: Correr el algoritmo BFS a partir del punto de partida. Se crea la estructura dfs_structs (92)	**O(V+E)**
P2: Con dicha estructura utilizamos la función bfs.pathTo para encontrar un camino entre los dos puntos.	**O(1)**
P3: Si existe un camino comenzamos a encontrar su información para presentarla al final del requerimiento.	**--**
P3.1: Creamos una lista para guardar el resultado (list_result) y creamos variables para guardar la distancia total recorrida y el número de puntos de encuentro (MTPs) en el camino. (95-98)	**O(1)**
P3.2: Por cada elemento del camino creamos un diccionario en el cual guardamos sus coordenadas, e identificador. También revisamos el próximo lobo y calculamos sus distancias, guardamos esta información en el diccionario y actualizamos la variable de distancia total.	**O(1)**
P3.3: Revisamos si el punto es un MTP, si lo es, miramos sus adyacentes y guardamos la informacion de los primeros y últimos lobos que pasaron por ese punto, de lo contrario, guardamos 'Desconocido'.	**O(1)**
P3.4: Guardamos el diccionario en list_result, tomamos sus primeros y últimos tres elementos y retornamos esa lista, el número de puntos de encuentro, la distancia total recorrida el path completo (será de utilidad al momento de crear el mapa)	**O(1)**
TOTAL	***O(V+E)***

Pruebas

Procesador	1,4 GHz Intel Core i5 de cuatro núcleos
Sistema operativo	MacOS Ventura 13.2.1
Punto inicial	m111p862_57p449
Punto final	m111p908_57p427

Entrada (pct)	**Tiempo (ms)**
1	9.32
5	29.77
10	77.58

20	150.32
30	972.77
50	1134.85
80	2199.87
100	2033.45



Análisis de complejidad

Como lo supusimos, en este caso la complejidad es claramente lineal y no tiene el comportamiento casi aleatorio del DFS. Los resultados del requerimiento son iguales para todas las muestras de datos, sin embargo, el tiempo de ejecución si cambia bastante, esto debido a que la ejecución del BFS cambia como O(V+E).

Requerimiento 3

En este requerimiento se busca encontrar las manadas de lobos a través de los componentes fuertemente conectados el grafo.

Entrada	La estructura de datos creada en la carga
Salidas	Una lista con las primeras y últimas 5 manadas encontradas en el grafo, el número de manadas distintas encontradas y un mapa con los vértices del grafo filtrados por manada.
Implementado (Sí/No)	Sí

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


    grafo = data_structs['wolf_tracks_graph']
    componentes_conectados  = scc.KosarajuSCC(grafo)
    num_manadas = componentes_conectados['components']
    mapa_manadas = mp.newMap(numelements=100)
    info_lobos = data_structs['wolf_individuals']
    for vertex in lt.iterator(mp.keySet(componentes_conectados['idscc'])):
        num_componente = me.getValue(mp.get(componentes_conectados['idscc'], vertex))
        if not mp.contains(mapa_manadas, num_componente):
            component_filter = mp.newMap(numelements=4)
            MTP_list = lt.newList(datastructure='ARRAY_LIST',
cmpfunction=standard_compare)
            seg_table = mp.newMap(numelements=50)
            list_todos = lt.newList(datastructure='ARRAY_LIST',
cmpfunction=standard_compare)
            lt.addLast(list_todos, vertex)
            if is_MTP(vertex):
                lt.addLast(MTP_list, vertex)
            else:
                id_lobo = seg_id_to_coords_id(vertex)[2]
                add_to_hash_table(seg_table, id_lobo, vertex)

            mp.put(component_filter, 'MTP',MTP_list)
            mp.put(component_filter,'seg', seg_table)
            mp.put(component_filter, 'todos', list_todos)
            mp.put(mapa_manadas,num_componente, component_filter )
        else:
            component_filter = me.getValue(mp.get(mapa_manadas, num_componente))
            MTP_list = me.getValue(mp.get(component_filter, 'MTP'))
            seg_table = me.getValue(mp.get(component_filter, 'seg'))
            todos = me.getValue(mp.get(component_filter, 'todos'))
            lt.addLast(todos, vertex)
```

```
            if is_MTP(vertex):
                lt.addLast(MTP_list, vertex)
            else:
                id_lobo = seg_id_to_coords_id(vertex)[2]
                add_to_hash_table(seg_table, id_lobo, vertex)

    list_result = lt.newList(datastructure='ARRAY_LIST')
    for manada in lt.iterator(mp.keySet(mapa_manadas)):
        info_manada = {}
        lobos_manada = lt.newList('ARRAY_LIST')
        filtro_MTPs = me.getValue(mp.get(mapa_manadas, manada))
        MTP_list = me.getValue(mp.get(filtro_MTPs, 'MTP'))
        seg_map = me.getValue(mp.get(filtro_MTPs, 'seg'))
        lista_todos = me.getValue(mp.get(filtro_MTPs, 'todos'))
        info_manada['Codigo manada'] = manada
        info_manada['Tamaño SCC'] = lt.size(lista_todos)
        info_manada['Primeros y últimos puntos'] =
first_last_n_elems_list(lista_todos, 3)['elements']
        info_manada['Num lobos'] = lt.size(mp.keySet(seg_map))
        primeros_ultimos_lobos = first_last_n_elems_list(mp.keySet(seg_map), 3)

        list_lats = lt.newList('ARRAY_LIST')
        list_longs = lt.newList('ARRAY_LIST')
        for node in lt.iterator(lista_todos):
            long_nod = id_to_coords(node)[1]
            lat_nod = id_to_coords(node)[0]
            lt.addLast(list_lats, lat_nod)
            lt.addLast(list_longs, long_nod)
        merg.sort(list_lats, sort_criteria_standard)
        merg.sort(list_longs, sort_criteria_standard)

        info_manada['Latitud mínima'] = lt.lastElement(list_lats)
        info_manada['Latitud máxima'] = lt.firstElement(list_lats)

        info_manada['Longitud mínima'] = lt.lastElement(list_longs)
        info_manada['Longitud máxima'] = lt.firstElement(list_longs)

        for lobo_seg in lt.iterator(primeros_ultimos_lobos):
            info_lobo = {}
            datos = me.getValue(mp.get(info_lobos, lobo_seg))
            info_lobo['Identificador'] = datos['individual-id']
            info_lobo['Taxonomía'] = datos['animal-taxon']
            info_lobo['Ciclo de vida'] = datos['animal-life-stage']
            info_lobo['Sexo'] = datos['animal-sex']
            info_lobo['Lugar de estudio'] = datos['study-site']
            lt.addLast(lobos_manada, info_lobo)
        info_manada['Primeros y últimos tres lobos en la manada'] = lobos_manada
        lt.addLast(list_result, info_manada)
```

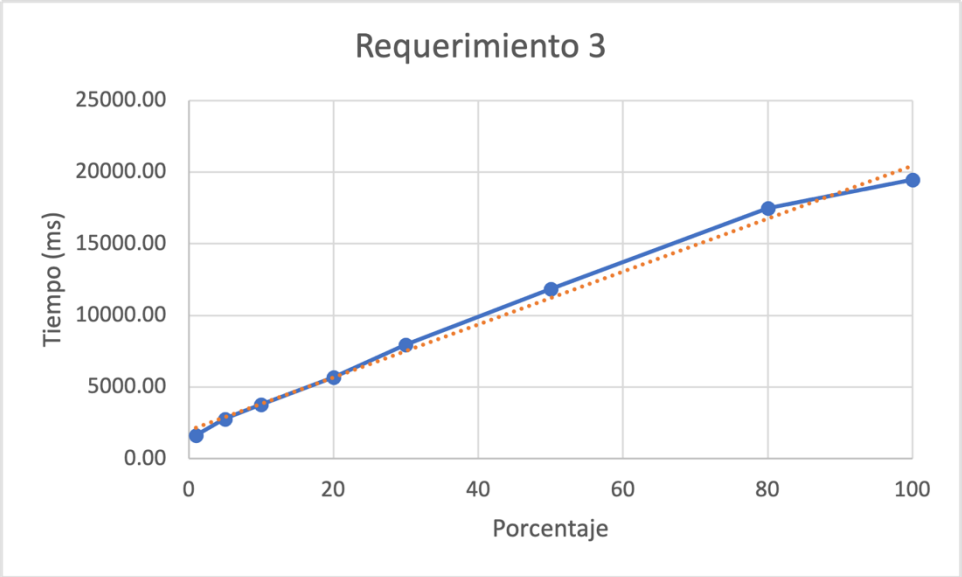
```
    merg.sort(list_result, sort_criteria_dominancia)

    return num_manadas, list_result, mapa_manadas
```

Pasos	**Complejidad**
P1: Se guarda la estructura que ejecuta el algoritmo de Kosaraju en la variable componentes_conectados. En esta estructura se guarda (principalmente) el número de componentes conectados y una tabla de hash con todos los vértices como llave y el componente conectado al que pertenecen como valor.	O(V+E)
P2: Comenzamos a acumular los componentes conectados en una sola tabla de hash, cuya llave es el número del componente conectado y cuyo valor es otra tabla de hash con listas con todos los vértices pertenecientes a dicho componente (mapa_manadas). Los elementos de esta tabla de hash son: una lista de nodos filtrados por MTP, una tabla de hash de puntos de seguimiento, filtrados a su vez por lobo, y una lista todos los elementos pertenecientes al componente. Recorremos la tabla de hash de componentes_conectados, revisamos si el componente conectado está en la tabla, si lo está añadimos el vértice a la lista con todos los elementos y revisamos si es o no MTP, luego añadimos el vértice a la lista correspondiente. Si no está, añadimos el diccionario para guardar los elementos de dicho componente.	O(V)
P3: Ya con la tabla de hash construida, recorremos sus llaves (los códigos de componente conectado), para sacar información al respecto.	O(V) [O(1)]
P3.1: Por cada componente conectado creamos un diccionario para guardar la información. Creamos una lista (lobos_manada) para guardar la información sobre cada lobo en el componente (subtablas). En el diccionario guardamos el código de la manada, el número de lobos en el componente, y los primeros y últimos puntos en el componente.	O(1)
P3.2: Creamos dos tablas para guardar las longitudes y latitudes de los puntos en el grafo. Iteramos sobre los nodos de la manada, guardamos sus latitudes y longitudes en la lista correspondiente y ordenamos utilizando el algoritmo mergesort las dos listas para obtener el máximo y mínimo elemento de cada una. Esto es, la latitud máxima y mínima de la manada. Añadimos esta información al diccionario de la manada.	O(V)
P3.3: Iteramos sobre la lista con los primeros y últimos lobos. Creamos un diccionario para cada lobo y basado en la tabla con la información de los lobos, lo llenamos. Añadimos este diccionario a la lista lobos_manada y guardmamos la lista como valor para una de las llaves del diccionario.	O(1)
P3.4: Insertamos el diccionario de la manada en una lista con el resultado.	O(1)
P4: Retornamos el mapa con las manadas, el número de manadas, y la lista de diccionarios con el resultado.	O(1)
Total	**O(V*V) [O(V)]**

Pruebas

Procesador	1,4 GHz Intel Core i5 de cuatro núcleos
Sistema operativo	MacOS Ventura 13.2.1

Entrada (pct)	Tiempo (ms)
1	1626.02
5	2791.72
10	3785.59
20	5676.75
30	7943.06
50	11838.52
80	17481.64
100	19464.08



Análisis de complejidad

Como supusimos en el análisis de complejidad teórico, este requerimiento tiene orden de crecimiento O(V). Esto tiene sentido pues el algoritmo utilizado para sacar los componentes fuertemente conectados tiene orden de complejidad O(V). El algoritmo NO es O(V*V) pues el número de manadas distintas se mantiene constante durante todo el requerimiento (incluso el 100% de los datos tiene 1200 manadas aproximadamente y el 1% tiene 1300 manadas aproximadamente). De ahí que el paso 2 no tiene complejidad O(V), tiene complejidad O(1) y el ciclo anidado para encontrar las latitudes y longitudes no tiene complejidad O(V*V)

Requerimiento 6

En este requerimiento se busca encontrar los lobos de cierto género más y menos distancia recorrió en un intervalo específico de tiempo.

Entrada	La estructura de datos creada en la carga, el tiempo inicial, el tiempo final y el género a consultar.
Salidas	La información del lobo que más recorrio, la información del lobo que menos recorrió, los primeros y últimos puntos de seguimiento del camino más largo de cada lobo, la distancia recorrida en cada camino, el número de aristas del camino y el número de vértices del camino.
Implementado (Sí/No)	Sí

```
def req_6(data_structs, initial_date_str, final_date_str, animal_sex):
    """
    Función que soluciona el requerimiento 6
    """
    # TODO: Realizar el requerimiento 6
    alternate_datastructs = new_data_structs()
    alternate_datastructs['wolf_tracks_graph'] = gr.newGraph(datastructure='ADJ_LIST',
directed = True, size= 200000)
    tree = alternate_datastructs['distance_tree']

    list_all_tracks = data_structs['all_tracks']

    individuals = data_structs['wolf_individuals']
    initial_date = str_to_datetime(initial_date_str)
    final_date = str_to_datetime(final_date_str)
    for track in lt.iterator(list_all_tracks):
        track_date = track['timestamp']
        track_id = track['individual-id']
        track_sex = me.getValue(mp.get(individuals, track_id))['animal-sex']
        if track_date >= initial_date and track_date <= final_date and track_sex ==
animal_sex:
            add_seg(alternate_datastructs,track, False)

    trace_paths(alternate_datastructs)

    #El camino más largo es simplemente es el camino completo que recorrió el lobo
durante el intervalo,
    # en este requerimiento podría hacerse sin grafos, pero para utilizar las
estructuras de datos, los utilizaremos
    graph = alternate_datastructs['wolf_tracks_graph']

    min_distance = om.minKey(alternate_datastructs['distance_tree'])
    max_distance = om.maxKey(alternate_datastructs['distance_tree'])

    min_id = me.getValue(om.get(tree, min_distance))
```

```
    max_id = me.getValue(om.get(tree, max_distance))

    min_wolf_info = me.getValue(mp.get(individuals, min_id))
    max_wolf_info = me.getValue(mp.get(individuals, max_id))

    min_wolf_info['Distancia recorrida'] = min_distance
    max_wolf_info['Distancia recorrida'] = max_distance

    #Lobo más largo
    tracks_lobo_max =
me.getValue(mp.get(alternate_datastructs['wolf_individual_track'], max_id))

    list_result_max = lt.newList('ARRAY_LIST')
    distance_max = 0
    num_vertices_max = 0
    num_edges_max = 0
    for index in range(1,lt.size(tracks_lobo_max)+1):
        track = lt.getElement(tracks_lobo_max, index)
        if gr.containsVertex(graph, track['animal-seg-id']):
            num_vertices_max += 1
            dict_vertex = {}
            dict_vertex['Vertex-id'] = track['animal-seg-id']
            vertex_longitude = track['location-long']
            vertex_latitude = track['location-lat']
            dict_vertex['Longitude'] = vertex_longitude
            dict_vertex['Latitude'] = vertex_latitude
            if gr.containsVertex(graph, id_MTP(vertex_longitude, vertex_latitude)):
                dict_vertex['Individuals in location'] = gr.degree(graph,
id_MTP(vertex_longitude, vertex_latitude))
            else:
                dict_vertex['Individuals in location'] = 1
            dict_vertex['Individual id'] = max_id
            if index != lt.size(tracks_lobo_max):
                next_track = lt.getElement(tracks_lobo_max, index + 1)
            if gr.getEdge(graph,track['animal-seg-id'], next_track['animal-seg-id'])
!= None:
                num_edges_max +=1
                distance_max += gr.getEdge(graph,track['animal-seg-id'],
next_track['animal-seg-id'])['weight']
            lt.addLast(list_result_max, dict_vertex)

    #Lobo más corto
    tracks_lobo_min =
me.getValue(mp.get(alternate_datastructs['wolf_individual_track'], min_id))

    list_result_min = lt.newList('ARRAY_LIST')
    distance_min = 0
    num_vertices_min = 0
```

```
    num_edges_min = 0
    for index in range(1,lt.size(tracks_lobo_min)+1):
        track = lt.getElement(tracks_lobo_min, index)
        if gr.containsVertex(graph, track['animal-seg-id']):
            num_vertices_min += 1
            dict_vertex = {}
            dict_vertex['Vertex-id'] = track['animal-seg-id']
            vertex_longitude = track['location-long']
            vertex_latitude = track['location-lat']
            dict_vertex['Longitude'] = vertex_longitude
            dict_vertex['Latitude'] = vertex_latitude
            if gr.containsVertex(graph, id_MTP(vertex_longitude, vertex_latitude)):
                dict_vertex['Individuals in location'] = gr.degree(graph,
id_MTP(vertex_longitude, vertex_latitude))
            else:
                dict_vertex['Individuals in location'] = 1
            dict_vertex['Individual id'] = min_id
            if index != lt.size(tracks_lobo_min):
                next_track = lt.getElement(tracks_lobo_min, index + 1)
            if gr.getEdge(graph,track['animal-seg-id'], next_track['animal-seg-id'])
!= None:
                num_edges_min +=1
                distance_min += gr.getEdge(graph,track['animal-seg-id'],
next_track['animal-seg-id'])['weight']
            lt.addLast(list_result_min, dict_vertex)
        min_wolf_info_list = lt.newList('ARRAY_LIST')
        lt.addLast(min_wolf_info_list, min_wolf_info)

        max_wolf_info_list = lt.newList('ARRAY_LIST')
        lt.addLast(max_wolf_info_list, max_wolf_info)
    return min_wolf_info_list, max_wolf_info_list, first_last_n_elems_list
(list_result_min, 3), first_last_n_elems_list (list_result_max, 3), distance_min,
num_edges_min, num_vertices_min, distance_max, num_edges_max, num_vertices_max
```

Pasos	**Complejidad**
P1: Se crea una estructura de datos alternativa para crear y guardar la información de los tracks que cumplan con las características dadas. (alternate datastructs)	O(1)
P2: Se convierten las fechas al formato datetime.	O(1)
P3: Se itera sobre la lista con todos los tracks (‘all_tracks’), en cada iteración, se revisa si el track cumple con las condiciones del problema, si si cumple, entonces se invoca la función add_seg sobre alternate_datastructs.	O(V)
P4: Se invoca la función trace_paths para conectar el grafo alternativo.	O(V*log(V)+E) [O(V)]

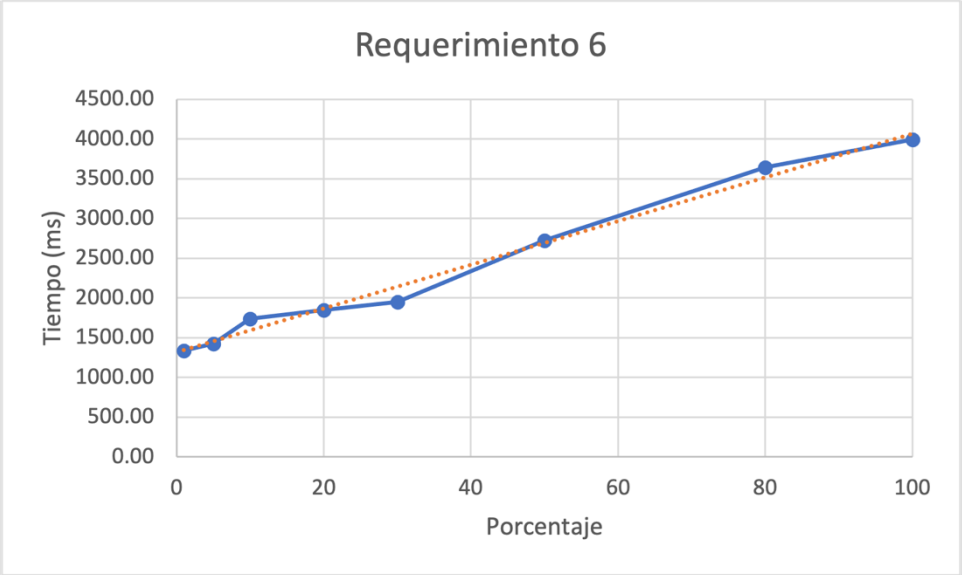
P5: Recuerde que en la función trace_paths, definimos un árbol. En este caso, tomamos el elemento mínimo del árbol y el elemento máximo del árbol. Los valores de estas parejas llave valor son los códigos del lobo que más y menos recorrió en el intervalo de tiempo.	O(1)
P6: De la información de los lobos sacamos los datos de cada uno de los dos individuos (mínimo y máximo). A esos diccionarios añadimos parejas llave valor con la distancia total recorrida por cada lobo.	O(1)
P7: El camino más largo recorrido por un lobo es simplemente todo el camino recorrido desde el comienzo del intervalo hasta el final. Por ende, recorreremos la lista con todos los tracks del lobo que más distancia recorrió, ('wolf_individual_tracks' en la estructura de datos alternativa)	
P7.1: Se inician contadores para número de aristas, de nodos, y la distancia total recorrida. Se consigue la lista con la información de tracks del lobo (tracks_lobo_max)	O(1)
P7.2: Se recorre la lista de tracks por el índice.	O(V)
P7.2.1: Se consigue el elemento en la posición, es decir, el i-ésimo track.	O(1)
P7.2.2: Se revisa si el vértice asociado al track está en el grafo.	O(1)
P7.2.3: Si lo está, se actualiza el contador de vértices y se crea un diccionario para guardar la información de dicho vértice. Se actualiza dicho diccionario con la información de posición y ID.	O(1)
P7.2.4: Para ver cuántos lobos pasan por ese punto, se revisa si el grafo contiene el MTP asociado al track, si sí lo contiene, eso significa que hay dos o más lobos en esa ubicación, y en tal caso, el número de lobos es el grado de el nodo MTP. De lo contrario el número de lobos en esa ubicación es 1. Se actualiza el diccionario con esta información	O(1)
P7.2.5: Si el índice es distinto al tamaño de la lista, se toma el siguiente elemento en la lista y se consigue el arco que los conecta. Si este arco existe (que siempre existirá), se actualiza el contador de arcos y se suma el peso del arco a la distancia total recorrida.	O(1)
P7.2.6: Se añade el diccionario a una lista para el lobo que mayor distancia recorrió	O(1)
P8: Se repite el paso 7, ahora para el lobo que menos distancia recorrió.	O(V)
Total	**O(V+V*log(V)+E) [O(V)]**

Pruebas

Procesador	1,4 GHz Intel Core i5 de cuatro núcleos
Sistema operativo	MacOS Ventura 13.2.1
Fecha inicial	2013-02-16 00:00
Fecha final	2014-10-23 23:59
Sexo	f

Entrada (pct)	**Tiempo (ms)**
1	1336.54

5	1426.79
10	1742.66
20	1847.78
30	1951.73
50	2726.28
80	3643.04
100	3993.04



Análisis de complejidad

Como supusimos en el análisis de complejidad, el requerimiento tiene complejidad O(V), y además, es sumamente eficiente, pues la pendiente de la gráfica es bastante baja. Esto se debe a que realmente, la iteración sobre los tracks, especialmente los tracks del lobo que menos recorrió, tienen complejidades muy bajas, pues estos lobos recorren poca distancia precisamente porque tienen pocos nodos en el intervalo especificado. La complejidad del requerimiento NO es O(V+V*log(V) +E), pues como explicamos en la carga de datos, en la práctica, la función trace_paths se comporta como O(V).

Requerimiento 7

En este requerimiento se busca encontrar las manadas de lobos reconocidas para tracks realizados en un intervalo de tiempo y temperaturas específicos pasados por parametro.

Entrada	La estructura de datos creada en la carga, el tiempo inicial, el tiempo final, la temperatura mínima y la temperatura máxima.
Salidas	Los componentes conectados de todos los vértices en dicho intervalo con su información. La ruta más larga de cada componente conectado. (Y el número de componentes conectados)
Implementado (Sí/No)	Sí

```
def req_7(data_structs, initial_date_str, final_date_str, min_temp, max_temp):
    """
    Función que soluciona el requerimiento 7
    """
    # TODO: Realizar el requerimiento 7
    alternate_data_structs = new_data_structs()
    alternate_data_structs['wolf_tracks_graph'] =
gr.newGraph(datastructure='ADJ_LIST', directed = True, size= 200000)

    initial_date = str_to_datetime(initial_date_str)
    final_date = str_to_datetime(final_date_str)

    individuals = data_structs['wolf_individuals']
    all_tracks = data_structs['all_tracks']

    alternate_data_structs['wolf_individuals'] = individuals

    for track in lt.iterator(all_tracks):
        track_timestamp = track['timestamp']
        track_wolf_temperature = float(track['external-temperature'])
        if track_timestamp >= initial_date and track_timestamp <= final_date and
track_wolf_temperature >= min_temp and track_wolf_temperature <= max_temp:
            add_seg(alternate_data_structs, track)

    trace_paths(alternate_data_structs)

    result_req_3 = req_3(alternate_data_structs)

    num_manadas = result_req_3[0]
    lista = result_req_3[1]
    mapa_manadas= result_req_3[2]
    vertex_components = result_req_3[3]
    #Para hallar el camino más largo, implementaremos un algoritmo que recorre todos
los vértices del componente en orden
    list_result = lt.newList('ARRAY_LIST')
    grafo = alternate_data_structs['wolf_tracks_graph']
```

```
    for info in lt.iterator(lista):
        codigo_manada = info['Codigo manada']
        size_manadas = info['Tamaño SCC']

        distance = 0

        LP_vertices = []
        LP_latitudes = []
        LP_longitudes = []
        list_manada = me.getValue(mp.get(me.getValue(mp.get(mapa_manadas,
codigo_manada)), 'todos'))
        LP_node = 0
        LP_edges = 0

        tabla_edges = mp.newMap()
        tabla_vertices = mp.newMap()

        for vertex in lt.iterator(list_manada):



            LP_vertices.append(vertex)
            LP_latitudes.append(id_to_coords(vertex)[0])
            LP_longitudes.append(id_to_coords(vertex)[1])
            LP_node += 1
            for edge in lt.iterator(gr.adjacentEdges(grafo, vertex)):
                va_comp = me.getValue(mp.get(vertex_components, edge['vertexA']))
                vb_comp = me.getValue(mp.get(vertex_components, edge['vertexB']))
                if va_comp == codigo_manada and vb_comp == codigo_manada:
                    if not mp.contains(tabla_edges, str(edge)):
                        mp.put(tabla_edges, str(edge), None)
                        LP_edges += 1
                        distance += edge['weight']


        max_long = max(LP_longitudes)
        max_lat = max(LP_latitudes)
        min_long = min(LP_longitudes)
        min_lat = min(LP_latitudes)

        dict_path = {}
        dict_path['Component ID'] = codigo_manada
        dict_path['SCC size'] = size_manadas
        dict_path['Minimum latitude'] = min_lat
        dict_path['Maximum latitude'] = max_lat
        dict_path['Minimum longitude'] = min_long
        dict_path['Maximum longitude'] = max_long
        dict_path['Nodes count'] = LP_node
```

```
        dict_path['Edges count'] = LP_edges
        dict_path['Distance'] = distance
        dict_path['Details'] = LP_vertices[0:5]

        lt.addLast(list_result, dict_path)
    return num_manadas, lista, mapa_manadas, first_last_n_elems_list(list_result, 5)
```

Pasos	**Complejidad**
P1: Se crea una estructura de datos alternativa para guardar los vértices que cumplan con la condiciones dadas. (alternate_data_structs)	**O(1)**
P2: Se convierten las fechas de str al formato datetime.	**O(1)**
P3: Se recorre la lista con todos los tracks de la estructura de datos original (‘all_tracks’), para cada track, se revisa si ocurrió en el intrevalo de fechas pasado por parámetro y en el intervalo de temperaturas. Si se cumple, se invoca la función add_seg para añadir el vértice a la estructrua de datos alterantiva.	**O(V)**
P4: Se usa la función trace_paths para conectar el grafo resultante en la estructura de datos alternativa.	**O(V+E)**
P5: Se corre el requerimiento 3 sobre la estructura de datos alternativa, este retorna el número de manadas, una lista con la información de las primeras manadas ordenadas por dominancia, una tabla de hash con la información de la manada y una talba de hash con todos los vértices y el componente conectado al que corresponden.	**O(V)**
P6: Al ser un componente fuertemente conectado, existe un camino continuo que contiene todas las aristas del componente, hallaremos entonces la información de ese camino.	**O(V*V)**
P6.1: Recorremos la lista con la información resumida de todas las manadas. A partir de esta lista sacamos el código de la manada y el número de vértices en el componente fuertemente conectado. Inicializamos un contador para la distancia total, el número de vértices, el número de aristas. Creamos tres listas, para guardar los vértices marcados, las latitudes, las longitudes. Finalmente creamos un diccionario para guardar la información del camino.	**O(1)**
P6.2: Recorremos la lista con todos los vértices en dicho componente conectado. Agregamos el vértice a la lista de vértices, su latitud y longitud a la lista respectiva y actualizamos el contador de vértices.	**O(V)[O(1)]**
P6.3: Recorremos las aristas adyacentes al vértice y revisamos si los dos vértices que conecta están en el mismo componente fuertemente conectado. Si es así, actualizamos el contador de aristas y agregamos a la distancia total el peso de la arista.	**O(1)**
P6.4: Encontramos la latitud y longitud máxima y mínima	**O(1)**
P6.5: En el diccionario de información del camino guardamos la información obtenida, eso es, las latitudes y longitudes máximas y mínimas, la lista con los vértices, la distancia total, el número de vértices y aristas, el código del componente fuertemente conectado y su tamaño	**O(1)**

P6.6: Añadimos el diccionario a una lista.	**O(1)**
P7: Retornamos las estructuras del requerimiento 3, además de la lista con la información de la ruta.	**O(1)**
Total	**O(V*V) [O(V)]**

Pruebas

Procesador	1,4 GHz Intel Core i5 de cuatro núcleos
Sistema operativo	MacOS Ventura 13.2.1
Fecha inicial	2012-11-28 00:00
Fecha final	2014-05-17 23:59
Temperatura mínima	-17.3
Temperatura máxima	9.7

Entrada (pct)	**Tiempo (ms)**
1	1688.94
5	2618.78
10	3584.50
20	5228.19
30	6926.79
50	10314.27
80	13822.57
100	17549.11

Análisis de complejidad

Como supusimos en el análisis de complejidad teórico, el requerimiento tiene un orden de crecimiento O(V). Nótese que el orden de crecimiento NO es realmente O(V*V) pues el número

de componentes fuertemente conectados NO crece como O(V), lo que crece verdaderamente es el tamaño de cada componente, por ende el ciclo del paso 6 no se realiza V-veces, lo que si crece es el número de vértices en cada componente, haciendo que la complejidad del paso 6 sea realmente O(V)