ANÁLISIS DEL RETO

Sebastián Palma, <s.palma@uniandes.edu.co>, 202222498.
Sara Leiva, <s.leivam@uniandes.edu.co>, 202220956.
Andrés Rodríguez, <a.rodiguezs@uniandes.edu.co>, 202222586.

Requerimiento 1

```
def req_1(data_structs, initial, destination):
    """
    Función que soluciona el requerimiento 1
    """
    grafo = data_structs["grafoDirigido"]
    containsInitial = gr.containsVertex(grafo, initial)
    containsDestination = gr.containsVertex(grafo, destination)
    if containsInitial and containsDestination:
        search = dfs.DepthFirstSearch(grafo, initial)
        haspath = dfs.hasPathTo(search, destination)
        if haspath:
            ipath = dfs.pathTo(search, destination)
            path = invertList(ipath)
            nG = getGatheringPoints(data_structs, path)
            nT = getTrackingPoints(data_structs, path)
            w = getWeight(grafo, path)
            return path, nG, nT, w
        else:
            return None
    else:
        return None
```

Entrada	Data_structs, vértice de inicio, vértice final
Salidas	La ruta de los nodos, los puntos de encuentro, los puntos de seguimiento y el total de la ruta
Implementado (Sí/No)	Si, implementado por Andrés Rodríguez

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Llamar al grafo dirigido con data_structs[“grafoDirigido”] y guardarlo en grafo	O(1)

Mirar si el punto de inicio, que esta guardado en initial, está dentro del grafo con gr.containsVertex y guardar el valor en containsInitial	O(1)
Mirar si el punto de destino, que esta guardado en destination, está dentro del grafo con gr.containsVertex y guardar el valor en containsDestination	O(1)
Evaluar la condicional	O(1)
Hacer un DFS en el grafo con el vértice inicial y guardar el valor en search	O(V + E)
Encontrar si existe un camino entre el punto de inicio y destino.	O(1)
Si haspathTo es igual a true, hacer pathTo con search y destinacion y guardarlo en ipath, obtener con una función externa los puntos de encuentro con data_structs y path y guardarlo en nG, obtener los puntos de seguimiento con una función externa con el data_structs y path y guardarlo en nT, también obtener el peso del recorrido con una función externa con el grafo y path y guardarlo en w .	O(1)
Si no, retornar None	O(1)
Si no, retornar None	O(1)
TOTAL	O(V + E)

Pruebas Realizadas

Procesadores	**Intel® Core™ i5-9300H CPU @2.40GHz**
Memoria RAM	8.0 GB
Sistema Operativo	Windows 10 Home Single – 64 bits

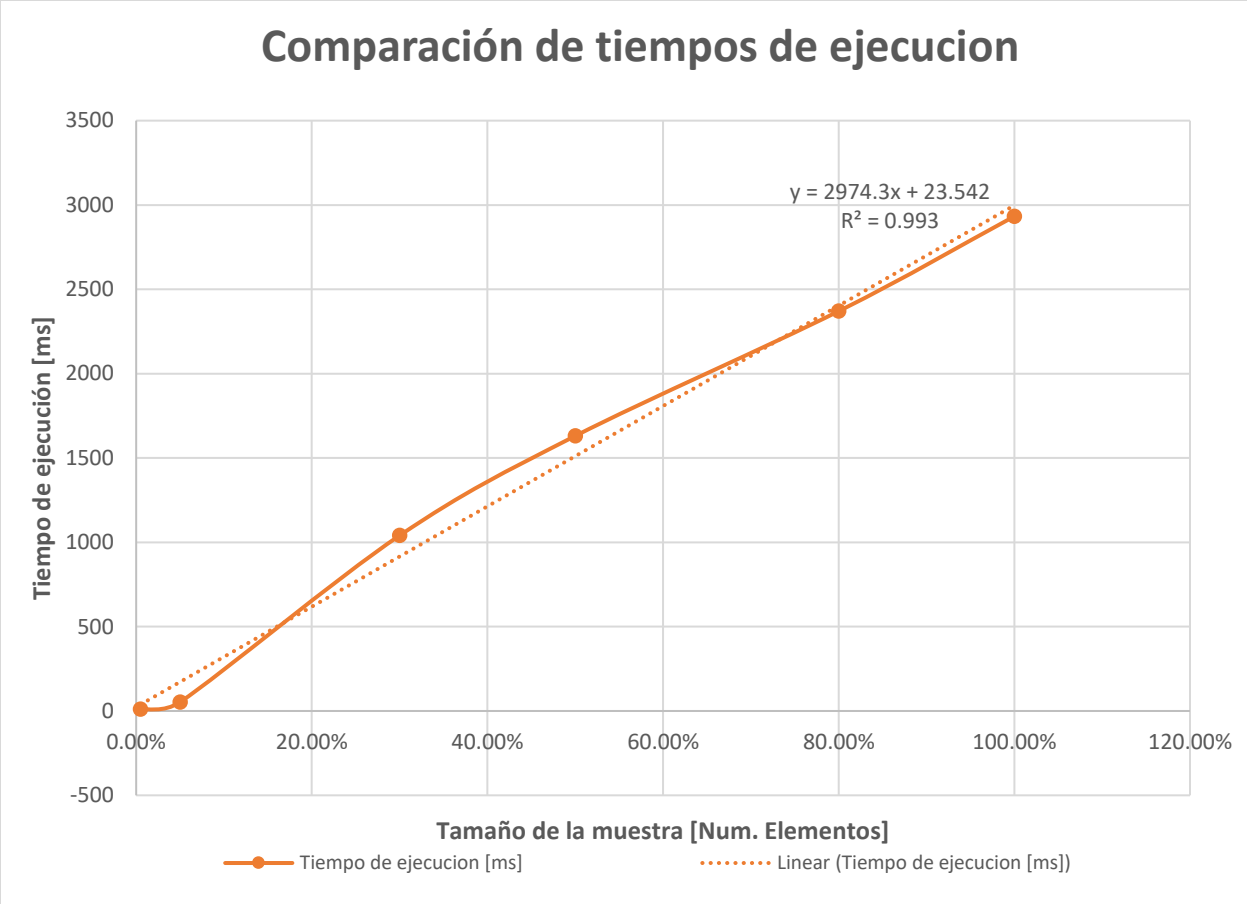
Entrada	**Tiempo (ms)**
small	10.94
5 pct	52.9
30 pct	1040.4
50 pct	1630.74
80 pct	2370.41
large	2932.56 – Implementado con BFS

Tablas de datos

Muestra	**Salida**	**Tiempo (ms)**

small	La ruta de los nodos, los puntos de encuentro, los puntos de seguimiento y el total de la ruta	10.94
5 pct	La ruta de los nodos, los puntos de encuentro, los puntos de seguimiento y el total de la ruta	52.9
30 pct	La ruta de los nodos, los puntos de encuentro, los puntos de seguimiento y el total de la ruta	1040.4
50 pct	La ruta de los nodos, los puntos de encuentro, los puntos de seguimiento y el total de la ruta	1630.74
80 pct	La ruta de los nodos, los puntos de encuentro, los puntos de seguimiento y el total de la ruta	2370.41
large	La ruta de los nodos, los puntos de encuentro, los puntos de seguimiento y el total de la ruta	2932.56 – Implementado con BFS

Graficas



Análisis

En este requerimiento se utilizó DFS para recorrer el grafo, con la intención de obtener el camino más barato entre dos puntos ingresados. El requerimiento tiene una complejidad temporal de O(V+E) dada por la complejidad del algoritmo DFS. Lo anterior nos evidencia el crecimiento proporcional del tiempo a medida que se incrementa el tamaño de arcos y arcos del grafo (el tamaño de la entrada de datos).

Requerimiento 2

```python
def req_2(data_structs, initial, destination):
    """
    Función que soluciona el requerimiento 2
    """
    grafo = data_structs["grafoDirigido"]
    containsInitial = gr.containsVertex(grafo, initial)
    containsDestination = gr.containsVertex(grafo, destination)
    if containsInitial and containsDestination:
        search = bfs.BreadhtFisrtSearch(grafo, initial)
        haspath = bfs.hasPathTo(search, destination)
        if haspath:
            ipath = bfs.pathTo(search, destination)
            path = invertList(ipath)
            nG = getGatheringPoints(data_structs, path)
            nT = getTrackingPoints(data_structs, path)
            w = getWeight(grafo, path)
            return path, nG, nT, w
        else:
            return None
    else:
        return None
```

Entrada	Data_structs, vértice de inicio, vértice final
Salidas	La ruta de los nodos, los puntos de encuentro, los puntos de seguimiento y el total de la ruta
Implementado (Sí/No)	Si, implementado por Andrés Rodríguez

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Llamar al grafo dirigido con data_structs[“grafoDirigido”] y guardarlo en grafo	O(1)
Mirar si el punto de inicio, que esta guardado en initial, está dentro del grafo con gr.containsVertex y guardar el valor en containsInitial	O(1)
Mirar si el punto de destino, que esta guardado en destination, está dentro del grafo con	O(1)

gr.containsVertex y guardar el valor en containsDestination	
Evaluar la condicional	O(1)
Hacer un BFS en el grafo con el vértice inicial y guardar el valor en search	O(V + E)
Encontrar si existe un camino entre el punto de inicio y destino.	O(1)
Si haspathTo es igual a true, hacer pathTo con search y destinacion y guardarlo en ipath, obtener con una función externa los puntos de encuentro con data_structs y path y guardarlo en nG, obtener los puntos de seguimiento con una función externa con el data_structs y path y guardarlo en nT, también obtener el peso del recorrido con una función externa con el grafo y path y guardarlo en w .	O(1)
Si no, retornar None	O(1)
Si no, retornar None	O(1)
TOTAL	O(V + E)

Pruebas Realizadas

Procesadores	**Intel® Core™ i5-9300H CPU @2.40GHz**
Memoria RAM	8.0 GB
Sistema Operativo	Windows 10 Home Single – 64 bits

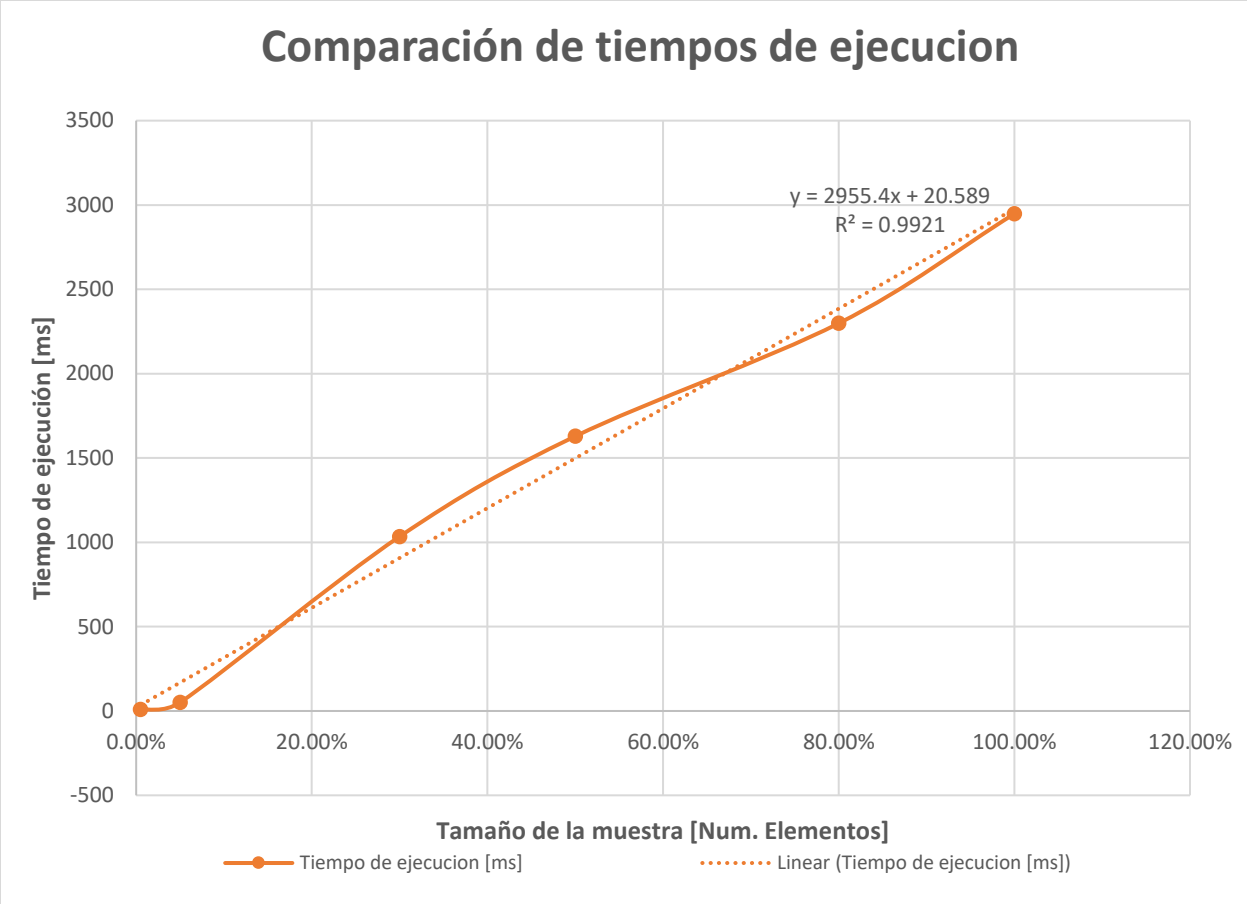
Entrada	**Tiempo (ms)**
small	10.03
5 pct	50.81
30 pct	1034.34
50 pct	1628.65
80 pct	2298.41
large	2947.85

Tablas de datos

Muestra	**Salida**	**Tiempo (ms)**
small	La ruta de los nodos, los puntos de encuentro, los puntos de seguimiento y el total de la ruta	10.03

5 pct	La ruta de los nodos, los puntos de encuentro, los puntos de seguimiento y el total de la ruta	50.81
30 pct	La ruta de los nodos, los puntos de encuentro, los puntos de seguimiento y el total de la ruta	1034.34
50 pct	La ruta de los nodos, los puntos de encuentro, los puntos de seguimiento y el total de la ruta	1628.65
80 pct	La ruta de los nodos, los puntos de encuentro, los puntos de seguimiento y el total de la ruta	2298.41
large	La ruta de los nodos, los puntos de encuentro, los puntos de seguimiento y el total de la ruta	2947.85

Graficas



Análisis

En este requerimiento se utilizó la estructura BFS para recorrer el grafo y encontrar el camino que existe de un punto inicial y final. De hecho, se utilizó BFS con la intención de obtener el camino más corto. Así mismo, este requerimiento tiene la misma complejidad que el requerimiento 1 gracias a que ambas estructuras comparten una complejidad de O(V+E). Luego, podemos ver que a medida que crece el número de arcos y vértices, crece el tiempo de ejecución de manera proporcional.

Requerimiento 4

```
def req_4(data_structs, origen_lon, origen_lat, dest_lon, dest_lat):
    """
    Función que soluciona el requerimiento 4
    """
    distance1 = 1000000
    distance2 = 1000000

    for punto_encuentro in lt.iterator(mp.keySet(data_structs["encuentro"])):

        if gr.containsVertex(data_structs["grafoDirigido"], punto_encuentro):
            punto_lon, punto_lat = noFormat(punto_encuentro)
```

```
            distance_origen = haversine(float(origen_lon), float(origen_lat),
float(punto_lon), float(punto_lat))
            distance_destino = haversine(float(punto_lon), float(punto_lat),
float(dest_lon), float(dest_lat))

            if distance_origen <= distance1:
                punto_cercano_origen = punto_encuentro
                distance1 = distance_origen
            if distance_destino <= distance2:
                punto_cercano_destino = punto_encuentro
                distance2 = distance_destino


    grafo_camino = djk.Dijkstra(data_structs["grafoDirigido"],
punto_cercano_origen)
    hasPath = djk.hasPathTo(grafo_camino, punto_cercano_destino)

    if hasPath == True:
        distTo = djk.distTo(grafo_camino, punto_cercano_destino)
        path = djk.pathTo(grafo_camino, punto_cercano_destino)
        trips_edges = lt.size(path)
        nodes = trips_edges + 1
        lista_puntos = lt.newList(datastructure = "ARRAY_LIST")

        for punto in lt.iterator(path):
            lt.addFirst(lista_puntos, punto)

        first_3 = lt.subList(lista_puntos, 1, 3)
        last_3 = lt.subList(lista_puntos, lt.size(lista_puntos)-2, 3)

        return punto_cercano_origen, punto_cercano_destino, round(distance1,3),
round(distance2,3),trips_edges, nodes, distTo, first_3, last_3

    else:
        return punto_cercano_origen, punto_cercano_destino, round(distance1,3),
round(distance2,3),  None, None, None, None, None
```

Entrada	Estructura de datos, punto de origen, punto de destino
Salidas	El punto más cercano al punto de origen y destino, las distancias entre estos puntos, las distancias de recorrido, los arcos del recorrido, los nodos, los primeros y últimos caminos.
Implementado (Sí/No)	Si, implementado por Sara Leiva

Estructura de datos usada	Para resolver este requerimiento se hizo uso del algoritmo Dijkstra, el cual permite calcular el camino más corto entre dos vértices a partir de un grafo dirigido con pesos positivos.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Definir dos variables que permiten comparar las distancias entre los puntos.	O(1)
Recorrer los puntos de encuentro.	O(n)
Si el punto de encuentro existe (está en el grafo), obtener la distancia al punto de origen y al punto de destino con la función haversine.	O(1)
Reemplazar los valores si se cumple las condiciones de estar más cerca al punto de origen y destino.	O(1)
Obtener el grafo con el camino más corto del punto más cercano al punto de origen con Dijkstra.	O(E log(V))
Preguntar si existe un camino desde el punto de origen cercano al punto destino cercano.	O(1)
Encontrar la distancia del camino más corto.	O(1)
Obtener el camino.	O(1)
Obtener el número de arcos (recorridos) del camino a partir del uso de lt.size()	O(1)
Obtener el número de nodos del camino al sumarle 1 al número de arcos.	O(1)
Organizar los puntos del camino a través de agregarlos a una lista.	O(n)
Obtener los primeros y últimos 3 recorridos del camino a través de sublistas.	O(3)
TOTAL	***O(E log(V))***

Pruebas Realizadas

Procesadores	**Intel® Core™ i7-1165G7 @2.80GHz**
Memoria RAM	15.6 GB
Sistema Operativo	Windows 11 Pro – 64 bits

Entrada	**Tiempo (ms)**
small	205.19
5 pct	1728.25
30 pct	55817.62

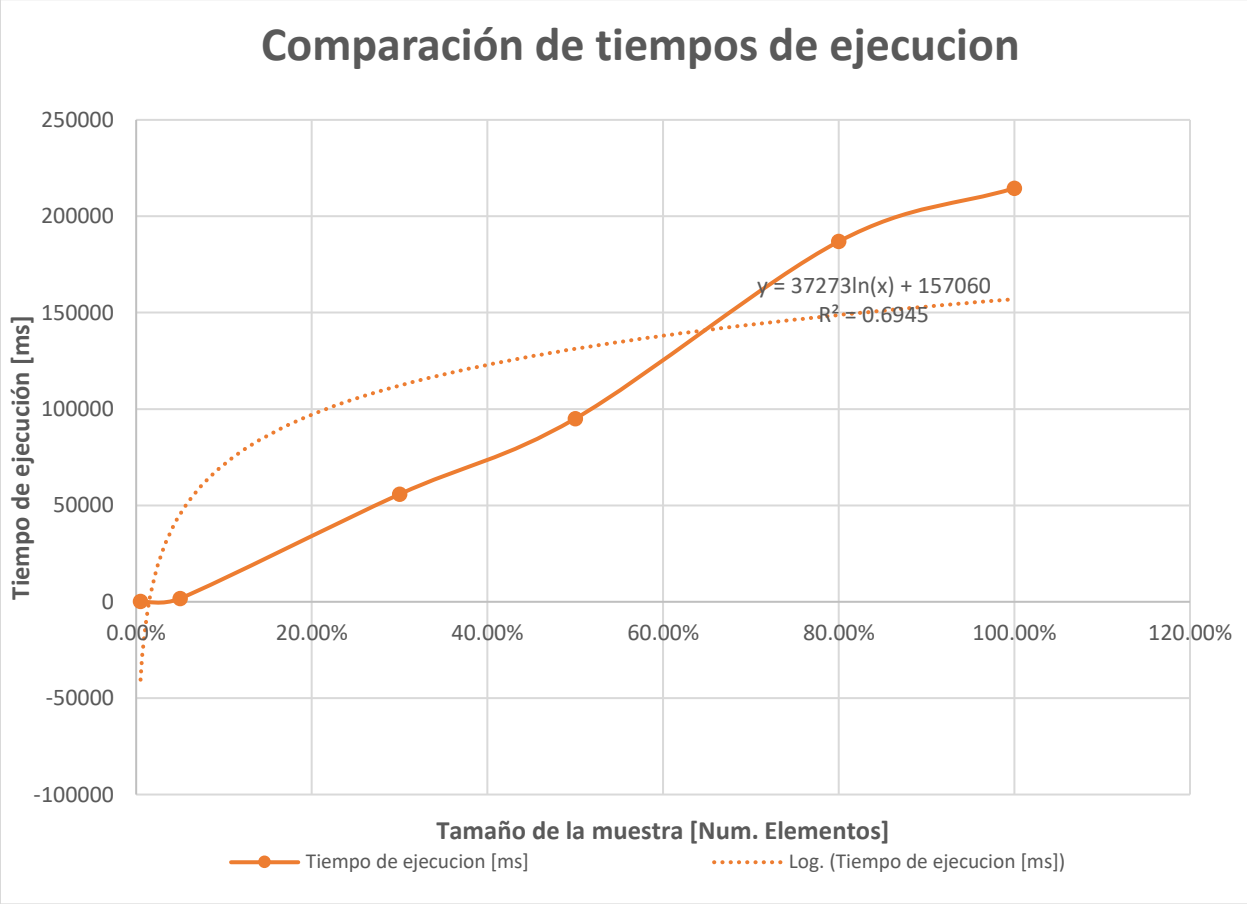
50 pct	95015.13
80 pct	186879.62
large	214537.87

Tablas de datos

Muestra	Salida	Tiempo (ms)
small	El punto más cercano al punto de origen y destino, las distancias entre estos puntos, las distancias de recorrido, los arcos del recorrido, los nodos, los primeros y últimos caminos.	205.19
5 pct	El punto más cercano al punto de origen y destino, las distancias entre estos puntos, las distancias de recorrido, los arcos del recorrido, los nodos, los primeros y últimos caminos.	1728.25
30 pct	El punto más cercano al punto de origen y destino, las distancias entre estos puntos, las distancias de recorrido, los arcos del recorrido, los nodos, los primeros y últimos caminos.	463184
50 pct	El punto más cercano al punto de origen y destino, las distancias entre estos puntos, las distancias de recorrido, los arcos del recorrido, los nodos, los primeros y últimos caminos.	95015.13
80 pct	El punto más cercano al punto de origen y destino, las distancias entre estos puntos, las distancias de recorrido, los arcos del	186879.62

	recorrido, los nodos, los primeros y últimos caminos.	
large	El punto más cercano al punto de origen y destino, las distancias entre estos puntos, las distancias de recorrido, los arcos del recorrido, los nodos, los primeros y últimos caminos.	214537.87

Graficas



Análisis

En este requerimiento se usó la estructura de Dijkstra para obtener el camino más corto entre dos puntos. A partir de este, obtenemos la distancia total del camino calculado (a partir de la estructura disto) y el camino (a través de la estructura edgeTo) que puede ser obtenido a partir de la función pathTo.(). A pesar de que este algoritmo nos retorna el camino que los lobos deben tomar para llegar más rápido a un punto determinado, podemos ver que al trabajar con una gran entrada de datos,

Dijkstra puede convertirse en un algoritmo ineficiente y demorado. Lo anterior se debe a su complejidad alta de O(E log(V)).

Requerimiento 5

Entrada	Puntos de encuentro minimos, distancia maxima, punto de encuentro de origen
Salidas	Diccionario con todas las posibles rutas desde el punto de encuentro de origen
Implementado (Sí/No)	Si

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Crea a los diccionarios, el valor acomulado, lista con los puntos recorridos	O(1)
Agregar el origen a la lista de puntos recorridos	O(1)
Llamar a la funcion recursiva GetPaths() hasta que se tengan los m puntos minimos de encuentro	O(E + V)^m
TOTAL	O(E + V)^m
Buscar los caminos mas cortos desde el origen con dijkstra	O(E + V)
Recorrer los puntos de encuentro en control	O(n)
Si el punto de encuentro no esta en la lista recorrida, revisar si su camino es menor junto al camino acomulado es menor a la distancia maxima	O(E + V)
Si es menor agregar el punto de encuentro a la lista y llamar a la funcion recursiva GetPaths()	O(E + V)^m
Si se revisaron todos los puntos y ninguno cumple las condicion anterior se revisa si el numero de puntos pasados es mayor o igual al minimo	O(1)
Si es menor se regresa los diccionarios con los caminos posibles	O(1)
Si es mayor o igual revisa los caminos entre los puntos de encuentro con dijkstra	O(E + V)

Por cada punto agregar el numero de animales en el mismo punto al diccionario	O(1)
Agrega todos los caminos posibles a una lista y luego la agrega al diccionario de posibles rutas	O(1)
Retorna el diccionario	O(1)

Pruebas Realizadas

Procesadores	**Intel® Core™ i5-9300H CPU @2.40GHz**
Memoria RAM	8.0 GB
Sistema Operativo	Windows 10 Home Single – 64 bits

Entrada	**Tiempo (ms)**
small	
5 pct	
10 pct	
20 pct	
30 pct	
50 pct	
80 pct	
large	

Tablas de datos

Muestra	**Salida**	**Tiempo (ms)**
small		
5 pct		
10 pct		
20 pct		
30 pct		
50 pct		
80 pct		
large		

Analizis

Como se ve en los pasos, gracias a que se utiliza una funcion recursiva se tiene que su complejidad es exponencial, dependiende del numero de iteraciones que se tengan que hacer, las cuales depende del numero minimo de puntos de encuentro

Requerimiento 6

```
def req_6(data_structs, fecha_inicial, fecha_final, sex):
    """
    Función que soluciona el requerimiento 6
    """
    # TODO: Realizar el requerimiento 6
    wolfs_ids = data_structs["wolfs"]

    distancia_menor = 1000000
    distancia_mayor = 0
    contador = 0

    for id in lt.iterator(mp.keySet(wolfs_ids)):
        distance_puntos = 0
        entry = mp.get(wolfs_ids, id)
        wolfs_info = me.getValue(entry)

        info = wolfs_info['info']['elements'][0]

        if info['animal-sex'] == sex:
            contador += 1
            mapa_fechas = wolfs_info['timestamp']
            datos_fechas = om.values(mapa_fechas, fecha_inicial, fecha_final)
            vertices = lt.newList(datastructure = "ARRAY_LIST")
            if lt.isEmpty(datos_fechas) == False:
                new_graph = gr.newGraph(datastructure = "ADJ_LIST", directed =
True, size = 50)
                for element in lt.iterator(datos_fechas):
                    punto_seguimiento = element['elements'][0]
                    gr.insertVertex(new_graph,
punto_seguimiento['punto_seguimiento'])
                    lt.addLast(vertices, punto_seguimiento['punto_seguimiento'])

                    if lt.size(vertices) > 1:
                        first_element = lt.getElement(vertices,
lt.size(vertices)-1)
                        gr.addEdge(new_graph, first_element,
punto_seguimiento['punto_seguimiento'], 0)
```

```
                        lon1,lat1 = noFormat(first_element)
                        lon2,lat2 =
noFormat(punto_seguimiento['punto_seguimiento'])
                        distance_puntos += haversine(float(lon1), float(lat1),
float(lon2), float(lat2))

                if distance_puntos <= distancia_menor:
                    distancia_menor = distance_puntos
                    id_menor = id
                    graph_menor = new_graph
                    vertices_menor = vertices

                if distance_puntos >= distancia_mayor:
                    distancia_mayor = distance_puntos
                    id_mayor = id
                    graph_mayor = new_graph
                    vertices_mayor = vertices


    arcos_mayor = lt.size(gr.edges(graph_mayor))
    nodos_mayor = arcos_mayor + 1
    if lt.size(vertices_mayor) > 6:
        first_3_mayor = lt.subList(vertices_mayor, 1, 3)
        last_3_mayor = lt.subList(vertices_mayor, lt.size(vertices_mayor)-2, 3)
    else:
        first_3_mayor = vertices_mayor
        last_3_mayor = None

    arcos_menor = lt.size(gr.edges(graph_menor))
    nodos_menor = arcos_menor + 1


    if lt.size(vertices_menor) > 6:
        first_3_menor = lt.subList(vertices_menor, 1, 3)
        last_3_menor = lt.subList(vertices_menor, lt.size(vertices_menor)-2, 3)
    else:
        first_3_menor = vertices_menor
        last_3_menor = None

total_distance_menor = 0

    entry_idmenor = mp.get(wolfs_ids, id_menor)
    menorwolf_info = me.getValue(entry_idmenor)
    map_menorwolf_info = menorwolf_info['timestamp']
    puntos_menor = om.valueSet(map_menorwolf_info)
```

```
    i = 0
    current = None

    for punto in lt.iterator(puntos_menor):
        current = punto['elements'][0]['punto']
        if i> 0:
            total_distance_menor  += CalculateDistance(current, menor_anterior)

        menor_anterior = current
        i += 1


    total_distance_mayor = 0
    entry_idmayor = mp.get(wolfs_ids, id_mayor)
    mayorwolf_info = me.getValue(entry_idmayor)
    map_mayorwolf_info = mayorwolf_info['timestamp']
    puntos1 = om.valueSet(map_mayorwolf_info)
    i = 0
    for punto in lt.iterator(puntos1):
        punto = punto['elements'][0]['punto']
        if i > 0:
            if punto_anterior != punto:
                distancia = CalculateDistance(punto_anterior, punto)
                total_distance_mayor += distancia

        punto_anterior = punto
        i += 1


    return id_menor, distancia_menor, nodos_menor, arcos_menor, first_3_menor,
last_3_menor, id_mayor, distancia_mayor, nodos_mayor, arcos_mayor, first_3_mayor,
last_3_mayor, contador, total_distance_menor, total_distance_mayor
```

Entrada	La estructura de datos, la fecha inicial y final y el sexo del animal.
Salidas	El id de los lobos pedidos, sus distancias recorridas, la cantidad de nodos y arcos en sus grafos de recorrido y los primeros y últimos 3 caminos del lobo.
Implementado (Sí/No)	Si

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Definir variables que serán usadas después.	O(1)
Recorrer los ids en el mapa que contiene la información de los lobos.	O(n)
Obtener el valor de la pareja (llave, valor) contenida en el mapa de los lobos. Así mismo, obtener la información (características) del lobo.	O(1)
Si cumple con la condición, obtener los puntos de seguimiento del lobo dentro del rango de fechas indicado con la función om.values()	O (1ogn + k) donde k es la cantidad de nodos en el rango.
Recorrer los elementos en el rango y obtener un punto de seguimiento.	O(d), donde d es la cantidad de fechas dentro del rango.
Insertar este punto en un nuevo grafo y añadir a una lista el vértice ingresado.	O(1)
Si la lista de los vértices es mayor a 1, tomar el elemento antes y recién ingresado y calcular su distancia con la función de haversine. Luego, sumarle esta distancia a la distancia total que será recorrida en este grafo.	O(1)
Comparar a través de condicionales y cambiar el valor de las variables.	O(1)
Obtener los arcos de los lobos que recorrieron un mayor y menor camino entre las fechas ingresadas por parámetro. Así mismo, obtener los nodos de aquellos lobos.	O(1)
Crear sublistas que guardarán los primero y últimos 3 recorridos de los lobos.	O(3)
Inicializar una variable en 0.	O(1)
Obtener la pareja <llave, valor> del mapa de los lobos.	O(1)
Obtener todos los puntos de seguimiento del lobo con om.valueSet().	O(1)
Recorrer cada punto de seguimiento del lobo.	O(n)
Obtener la distancia entre los puntos y sumarla a la distancia total del lobo.	O(1)
TOTAL	***O(n*d)***

Pruebas Realizadas

Procesadores	**Intel® Core™ i5-9300H CPU @2.40GHz**
Memoria RAM	8.0 GB
Sistema Operativo	Windows 10 Home Single – 64 bits

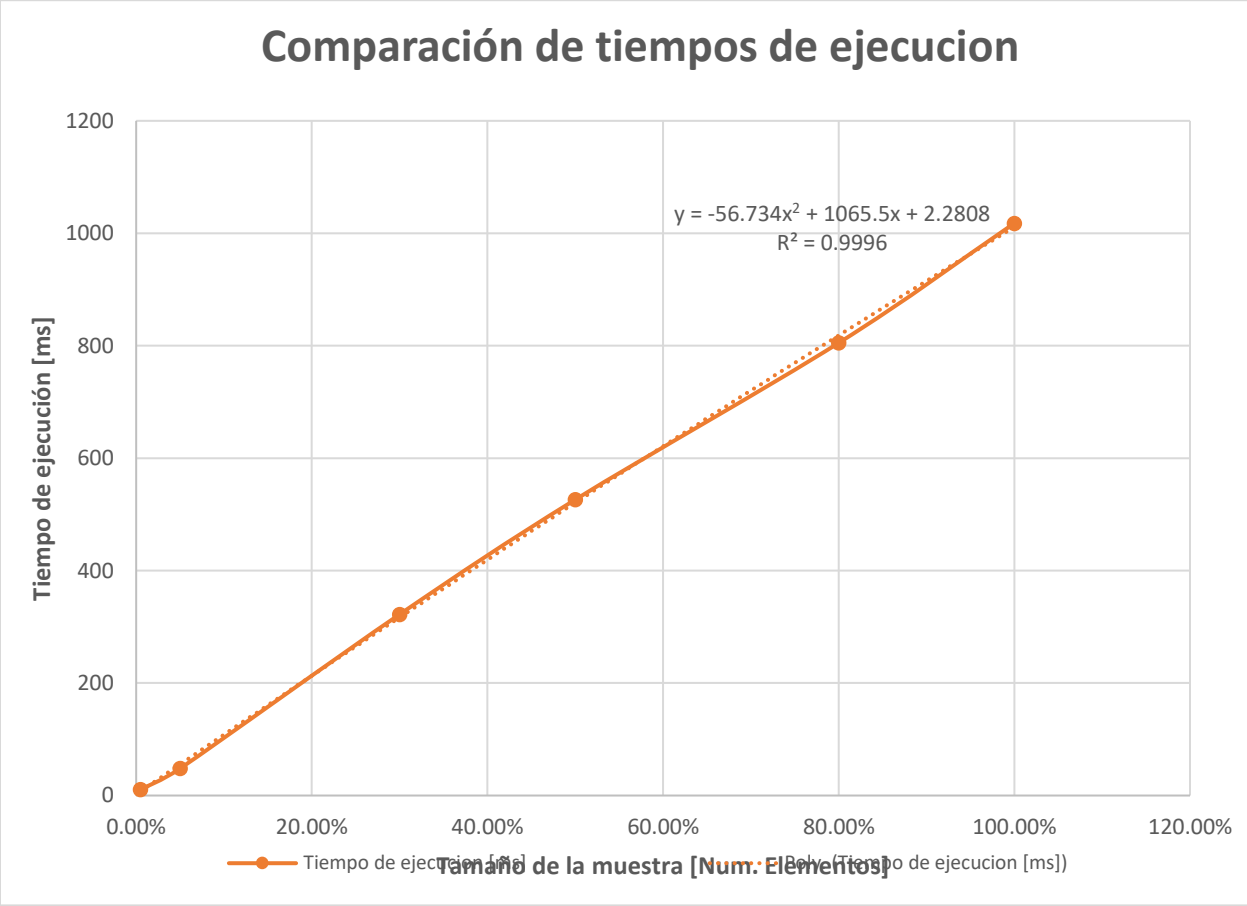
Entrada	Tiempo (ms)
small	10.81
5 pct	48.08
30 pct	322.23
50 pct	526.24
80 pct	805.01
large	1017.85

Tablas de datos

Muestra	Salida	Tiempo (ms)
small	El id de los lobos pedidos, sus distancias recorridas, la cantidad de nodos y arcos en sus grafos de recorrido y los primeros y últimos 3 caminos del lobo.	10.81
5 pct	El id de los lobos pedidos, sus distancias recorridas, la cantidad de nodos y arcos en sus grafos de recorrido y los primeros y últimos 3 caminos del lobo.	48.08
30 pct	El id de los lobos pedidos, sus distancias recorridas, la cantidad de nodos y arcos en sus grafos de recorrido y los primeros y últimos 3 caminos del lobo.	322.23
50 pct	El id de los lobos pedidos, sus distancias recorridas, la cantidad de nodos y arcos en sus grafos de recorrido y los primeros y últimos 3 caminos del lobo.	526.24
80 pct	El id de los lobos pedidos, sus distancias recorridas, la cantidad de nodos y arcos en sus grafos de recorrido y los primeros y últimos 3 caminos del lobo.	805.01
large	El id de los lobos pedidos, sus distancias recorridas, la	1017.85

	cantidad de nodos y arcos en sus grafos de recorrido y los primeros y últimos 3 caminos del lobo.	

Graficas



Análisis

Este requerimiento se soluciono a partir de actualizar la información que cumplía con una condicional determinada, lo que permitió que su complejidad fuera de O(n * d) al tener que recorrer dos veces la información en el grafo. Por esto, no fue necesario utilizar estructuras externas de los grafos y el tiempo de ejecución vs el tamaño del archivo pudo verse acotado por medio de una función cuadrática. No obstante, podemos ver que la gráfica es muy parecida a una gráfica lineal, pero no podemos decir que su complejidad es de O(n) gracias a que se hacen un recorrido dentro de otro.

Requerimiento 7

```
def req_7(data_structs, lowtime, hightime, lowtemp, hightemp):
    """
    Función que soluciona el requerimiento 7
    """
    new_graph_info(data_structs["temp"])
    graphInfo = data_structs["temp"]
    wolfs = data_structs["wolfs"]
    lowsuf = datetime.datetime.strptime("00:00", '%H:%M').time()
    highsuf = datetime.datetime.strptime("23:59", '%H:%M').time()
    lowtime = datetime.datetime.strptime(lowtime, '%Y-%m-%d')
    lowdate = lowtime.date()
    lowtime = lowtime.combine(lowdate, lowsuf)
    hightime = datetime.datetime.strptime(hightime, '%Y-%m-%d')
    highdate = hightime.date()
    hightime = hightime.combine(highdate, highsuf)
```

```
    ################### GET THE USABLE VALUES IN THE TIME AND TEMPERATURE ###################
    for wolf_id in lt.iterator(mp.keySet(wolfs)):
        entry = mp.get(wolfs, wolf_id)
        wolfs_info = me.getValue(entry)
        times = om.values(wolfs_info['timestamp'], lowtime, hightime)
        for time in lt.iterator(times):
            for info in lt.iterator(time):
                temp = float(info["temperature"])
                if (temp ≥ float(lowtemp)) and (temp ≤ float(hightemp)):
                    tracking = {"point": info["punto"],
                                "tracking": info["punto_seguimiento"],
                                "id" : info["id"]}
                    exists = mp.contains(graphInfo['seguimiento'], info["id"])
                    if exists:
                        value = mp.get(graphInfo['seguimiento'], info["id"])
                        track = me.getValue(value)
                        lt.addLast(track, tracking)
                    else:
                        lista = lt.newList(datastructure = "SINGLE_LINKED")
                        lt.addLast(lista, tracking)
                        mp.put(graphInfo['seguimiento'], info["id"], lista)
                    lt.addLast(graphInfo['lista_seguimientos'], info["punto_seguimiento"])
                    punto = info["punto"]
                    lon = punto[1]
                    lat = punto[0]
                    lon = str(lon).replace("-", "m")
                    lat = str(lat).replace("-", "m")
                    lon = lon.replace(".", "p")
                    lat = lat.replace(".", "p")
                    encuentro = lon + "_" + lat
                    contains = mp.contains(graphInfo["encuentro"], encuentro)
```

```
    if contains:
        entry_encuentro = mp.get(graphInfo["encuentro"], encuentro)
        lista_seguimiento = me.getValue(entry_encuentro)
        if not lt.isPresent(lista_seguimiento, info["punto_seguimiento"]):
            lt.addLast(lista_seguimiento, info["punto_seguimiento"])
    else:
        seguimientos=lt.newList(datastructure="ARRAY_LIST", cmpfunction=compareTracks)
        lt.addLast(seguimientos, info["punto_seguimiento"])
        mp.put(graphInfo["encuentro"], encuentro, seguimientos)
```

```
################### INSERT THE GATHERING POINTS TO THE GRAPH ###################
for gathering_point in lt.iterator(mp.keySet(graphInfo["encuentro"])):
    entry = mp.get(graphInfo["encuentro"], gathering_point)
    values = me.getValue(entry)
    if lt.size(values) ≥ 2:
        if not gr.containsVertex(graphInfo['temp_graph'], gathering_point):
            gr.insertVertex(graphInfo['temp_graph'], gathering_point)
            lt.addLast(graphInfo["lista_encuentros"], gathering_point)
    for tracking_point in lt.iterator(values):
        if not gr.containsVertex(graphInfo['temp_graph'], tracking_point):
            gr.insertVertex(graphInfo['temp_graph'], tracking_point)
        if gr.containsVertex(graphInfo['temp_graph'], gathering_point):
            edge1 = gr.getEdge(graphInfo['temp_graph'], gathering_point, tracking_point)
            if edge1 is None:
                gr.addEdge(graphInfo['temp_graph'], gathering_point, tracking_point, 0)
            edge2 = gr.getEdge(graphInfo['temp_graph'], tracking_point, gathering_point)
            if edge2 is None:
                gr.addEdge(graphInfo['temp_graph'], tracking_point, gathering_point, 0)
```

```
################### INSERT THE TRACKING POINTS EDGES TO THE NEW GRAPH ###################
for wolf in lt.iterator(mp.keySet(graphInfo["seguimiento"])):
    current = None
    for seguimiento in lt.iterator(me.getValue(mp.get(graphInfo["seguimiento"], wolf))):
        if current == None:
            current = seguimiento
        else:
            if current["tracking"] ≠ seguimiento["tracking"]:
                d = CalculateDistance(current['point'], seguimiento['point'])
                gr.addEdge(graphInfo["temp_graph"], current['tracking'], seguimiento['tracking'], d)
            current = seguimiento

################### CALCULATE SCC REQUIREMENT ###################
kosaraju = scc.KosarajuSCC(graphInfo["temp_graph"])
stcoco = kosaraju["idscc"]
```

```
  ################### GET INFO FOR ANSWER ###################
  sccelements = mp.newMap(numelements=1000, maptype="PROBING", loadfactor=0.5)
  organizer = lt.newList(datastructure="SINGLE_LINKED")
  longest = lt.newList(datastructure="SINGLE_LINKED")
  for point in lt.iterator(mp.keySet(stcoco)):
      e = mp.get(stcoco, point)
      sccid = me.getValue(e)
      inorder = mp.contains(sccelements, sccid)
      if inorder:
          scclist = me.getValue(mp.get(sccelements, sccid))
          lt.addLast(scclist, point)
      else:
          scclist = lt.newList(datastructure="SINGLE_LINKED")
          lt.addLast(scclist, point)
          mp.put(sccelements, sccid, scclist)


  for element in lt.iterator(mp.keySet(sccelements)):
      l = me.getValue(mp.get(sccelements, element))
      newgraph = gr.newGraph(datastructure="ADJ_LIST", directed=True)
      minmax = getMinMax(l)
      i = {"sccid": element,
           "list": l,
           "len": lt.size(l),
           "latInfo": minmax[1],
           "lonInfo": minmax[0],
           "gatNum": getGatheringPoints(graphInfo, l),
           "subgraph": create_sub_graph(graphInfo["temp_graph"], newgraph, l)}
      lt.addLast(organizer, i)
```

```
    for component in lt.iterator(organizer):
        long = dfo.DepthFirstOrder(component["subgraph"])
        path = stackToList(long['post'])
        longpath, weight = getLongestPath(component["subgraph"], path)
        minmax = getMinMax(longpath)
        j = {"SCCid": component["sccid"],
             "size": component["len"],
             "path": longpath,
             "len": lt.size(longpath),
             "latInfo": minmax[1],
             "lonInfo": minmax[0],
             "node": lt.size(longpath),
             "edges": lt.size(longpath) - 1,
             "weight": weight}
        lt.addLast(longest, j)
    merg.sort(organizer, compareNum)
    merg.sort(longest, compareWeight)
    return kosaraju, organizer, longest
```

Entrada	Data_structs, menor tiempo, mayor tiempo, menor temperatura, mayor temperatura
Salidas	Kosajaru del grafo y 2 listas,
Implementado (Sí/No)	Si, implementado por Andrés Rodríguez

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Crea un nuevo grafo con ulna function externa y data_structs[“temp”]	O (1)
Guarda data_structs[“temp”] en graphInfo	O (1)
Guarda data_structs[“wolfs”] en wolfs	O (1)
Agrega un formato de hora y lo guarda en lowsuf	O (1)
Agrega un formato de hora y lo guarda en highsuf	O (1)
Agrega un formato de hora y lo guarda en lowtime	O (1)
Hace un .date con lowtime, que entra por parámetro, y lo guarda en lowdate	O (1)
Hace un lowtine.combine con lowdate y lowsuf y lo guarda en low time	O (1)
Agrega un formato de fecha y lo guarda en hightime	O (1)
Hace hightime.date y lo guarda en highdate	O (1)
Hace un highdate.combine con highdate y highsuf	O (1)
Iterar wolf_id sobre una lista con las llaves de wolfs con lt.iterator y mp.keySet	O (n)
Buscar una pareja con la llave wolf_if en el mapa de wolfs con mp.get	O (n)

Obtener el valor de la pareja obtenida	O (1)
Obtener todos los valores de wolfs_info[“timestamp”] entre lowtime y hightime con om.values y guardarlo en times	O (1ogn + k) donde k es la cantidad de nodos en el rango.
Iterar time sobre times con lt.iterator	O (n)
Iterar info sobre time con lt.iterator	O (n)
Transformar info[“temperature”] a float y guardarlo en temp	O (1)
Si temp es mayor o igual al float de low temp y temp es menor o igual al float de hightemp, hacer un diccionario donde point = info [“punto”], tracking = info[“punto_seguimiento”], id = [“id”] y guardar todo esto es un diccionario llamado tracking	O (n)
Mirar si dentro del mapa graphInfo[“seguimiento”] esta info[“id”] con mp.contains y guardar el valor en exists	O (n)
Si exists es igual a true, obtener una pareja con la llave info[“id”] del mapa graphInfo[“seguimiento”] y guardarla en value	O (n)
Obtener solo el valor de la pareja guardada en value con me.getValue	O (1)
Agregar tracking al final de track	O (1)
Si no, crear una lista con lt.newList y guardarla en lista, añadir tracking al final de lista, y meter dentro de graphInfo[“seguimiento”] una pareja con llave info[“id”] y de valor lista	O (1)
Punto es igual a info[“punto”]	O (1)
Lon = punto [1]	O (1)
Lat = punto [0]	O (1)
Cambiar lon a string y usar un. remplace para reemplazar todas las – por m y guardarlo en lon	O (1)
Cambiar lat a string y usar un. remplace para reemplazar todas las – por m y guardarlo en lat	O (1)
Reemplazar en lon con un .remplace los . por p y guardarlo en lon	O (1)
Reemplazar en latcon un .remplace los . por p y guardarlo en lat	O (1)
Unir lon más un – más lat y guardarlo en encuentro	O (1)
Mirar si dentro del mapa graphInfo[“encuentro”] este encuentro con mp.contains y guardar el valor en contains	O (n)
Si contains es igual a true, obtener una pareja de llave encuentro en el mapa graphInfo[“encuentro”] y guardarla en entry_encuentro	O (n)

Obtener el valor de la pareja anterior con me.getValue y guardarlo en lista_seguimiento	O (1)
Con un lt.isPresent, se mira si dentro de lista de seguimiento esta info[“punto_seguimiento”], si no esta se agrega info[“punto_seguimiento”] al final de lista_seguimiento	O (n)
Si no, crear una nueva lista con lt.newList y guardarla en seguimientos, agregar info[“punto_seguimiento”] al final de seguimientos con lt.addLast, y meter una pareja en el mapa graphInfo[“encuentro”] con llave encuentro y valor seguimientos	O (1)
Iterar gathering_point sobre una lista con las llaves de graphInfo[“encuentro”] con lt.iterator y mp.keySet	O (n)
Obtener una pareja dentro del mapa graphInfo[“encuentro”] que tenga una llave igual a gathering_point con mp.get y guardala en entry	O (n)
Obtener el valor de la pareja anterior y guardarlo en values	O (1)
Si el size, sacado con lt.size, de values es mayor o igual a 2, si no está presente gathering_point dentro del grafo graphInfo[“temp_graph”], entonces insertar gathering_point dentro del grafo graphInfo[“temp_graph”] con gr.insertVertex y agragar gathering_point al final de graphInfo[“lista_encuentro”] con lt.addLast	O (n)
Iterar tracking_point sobre values con lt.iterator	O (n)
Si no está presente tracking_point dentro del grafo graphInfo[“temp_graph”], entonces insertar tracking_point dentro del grafo graphInfo[“temp_graph”] con gr.insertVertex	O (n)
Si está presente tracking_point dentro del grafo graphInfo[“temp_graph”], entonces retornar un arco asociado entre gathering_point y tracking_point del grafo graphInfo[“temp_graph”] con gr.getEdge y guardarlo en edge1	O (n)
Si edge1 es None, agregar un arco entre gathering_point y tracking_point con peso 0 en el grafo graphInfo[“temp_graph”] con gr.addEdge	O (n)
Retornar un arco asociado entre tracking_point y gathering_point del grafo graphInfo[“temp_grafph”] con gr.getEdge y guardarlo en edge2	O (n)
Si edge2 es igual a None, agregar un arco entre tracking_point y gathering_point con peso 0 en el grafo graphInfo[“temp_graph”] con gr.addEdge	O (n)

Iterar wolf sobre una lista con las llaves de graphInfo[“seguimiento”] con lt.iterator y mp.keySet	O (n)
Inicializar current = None	O (1)
Iterar seguimiento Iterar wolf sobre el valor de la pareja obtenida con la llave wolf en el grafo graphInfo[“seguimiento”] con mp.get y me.getValue	O (n)
Si current es igual a None, current es igual a seguimiento	O (1)
Si no, si current[“tracking”] es diferente de seguimiento[“traking”], se calcula la distancia entre current[“point”] y seguimiento[“point”] y se guarda en d, luego agregar un arco entre current[“tracking”] y seguimiento[“traking”] con peso d en el grafo graphInfo[“temp_graph”]	O (n)
current = seguimiento	O (1)
Sacar el kosaraju del grafo graphInfo[“temp_graph”] con scc.KosarajuSCC y guardarlo en kosarau	O (V + E)
Stcoco = kosaraju[“idscc”]	O (1)
Crear un nuevo mapa con mp.newMap y guardarlo en sccelementes	O (1)
Crear una lista con lt.newList y guardarla en organizer	O (1)
Crear una lista con lt.newList y guardarla en longest	O (1)
Iterar point sobre una lista de las llaves de stcoco con mp.keySet y lt.iterator	O (n)
Retornar una pareja que la llave sea igual a point dentro de stcoco con mp.get y guardarlo en e	O (n)
Sacar el valor de esa pareja con me.getValue y guardarlo en sccid	O (1)
Mirar si el valor sccid se encuentra dentro de sccelementes con un mp.contains y guardarlo en inorder	O (n)
Si inorner es true, retornar el valor de la pareja que la llave es igual a sccid dentro de sccelementes y guardarla en scclist	O (n)
Meter point al final de scclist con lt.addLast	O (1)
Si no, crear una nueva lista con lt.newList y guardarla en scclist, meter al final de scclist con lt.addLast y meter dentro de sccelements una pareja donde la llave es sccide y el valor es scclist	O (1)
Iterar element sobre una lista de las llaves de sccelements con mp.keySet y lt.iterator	O (n)

Retornar el valor de la pareja que es igual a la llave element dentro de sccelementes y guardarla en scclist y guardarlo en l	O (n)
Crear un nuevo grafo y guardarlo dentro de newgraph y guardarlo en minmax	O (1)
Retornar las longitudes y latitudes máximas y mínimas con una funcion externa de l y guardar en minmax	O (n)
Inicializar i como un diccionario que tiene “sccid” = element, “list” = l, “len” = lt,size(l), “latInfo” = minmax[1]. “lonInfo” = minmax[0], “gatNum” = getGatheringPoints(graphInfo, l), “subgraph” = create_sub_graph(graphInfo[“temp_graph”], newgraph, l)	O (n)
Meter i al final de organizer con lt.addLast	O (1)
Iterar component sobre organizer con lt.iterator	O (n)
Hacer un DFS del grafo component[“subgraph”] y guardarlo en long	O(V + E)
Hacer stackToList de long[“post”] y guardarlo en path	O (n)
Hacer getLongestPath de component[“subgraph”] y path y guardarlo en longpath y weight	O (n)
Hacer getMinMx de longpath y guardarlo en minmax	O (n)
Inicializar j como un diccionario que tiene “SCCid” = component[“sccid”], “size” = component[“len”], “path” = longpath, “len” = lt.size(longpath), “latInfo” = minmax[1]. “lonInfo” = minmax[0], “node” = lt.size(longpath), “edges” = lt.size(longpath) - 1, “weight” = weight	O (1)
Meter j al final de longest con lt.addLast	
Hacer merg.sort de organizer con compareNum	O (n*Log(n))
Hacer merg.sort de longest con compareWeight	O (n*Log(n))
Retornar kosaraju, organizer, longest	O (1)
TOTAL	O(V + E)

Pruebas Realizadas

Procesadores	**Intel® Core™ i5-9300H CPU @2.40GHz**
Memoria RAM	8.0 GB
Sistema Operativo	Windows 10 Home Single – 64 bits

Entrada	**Tiempo (ms)**
small	1060.55
5 pct	3279.84
10 pct	5882.69
20 pct	12589.89

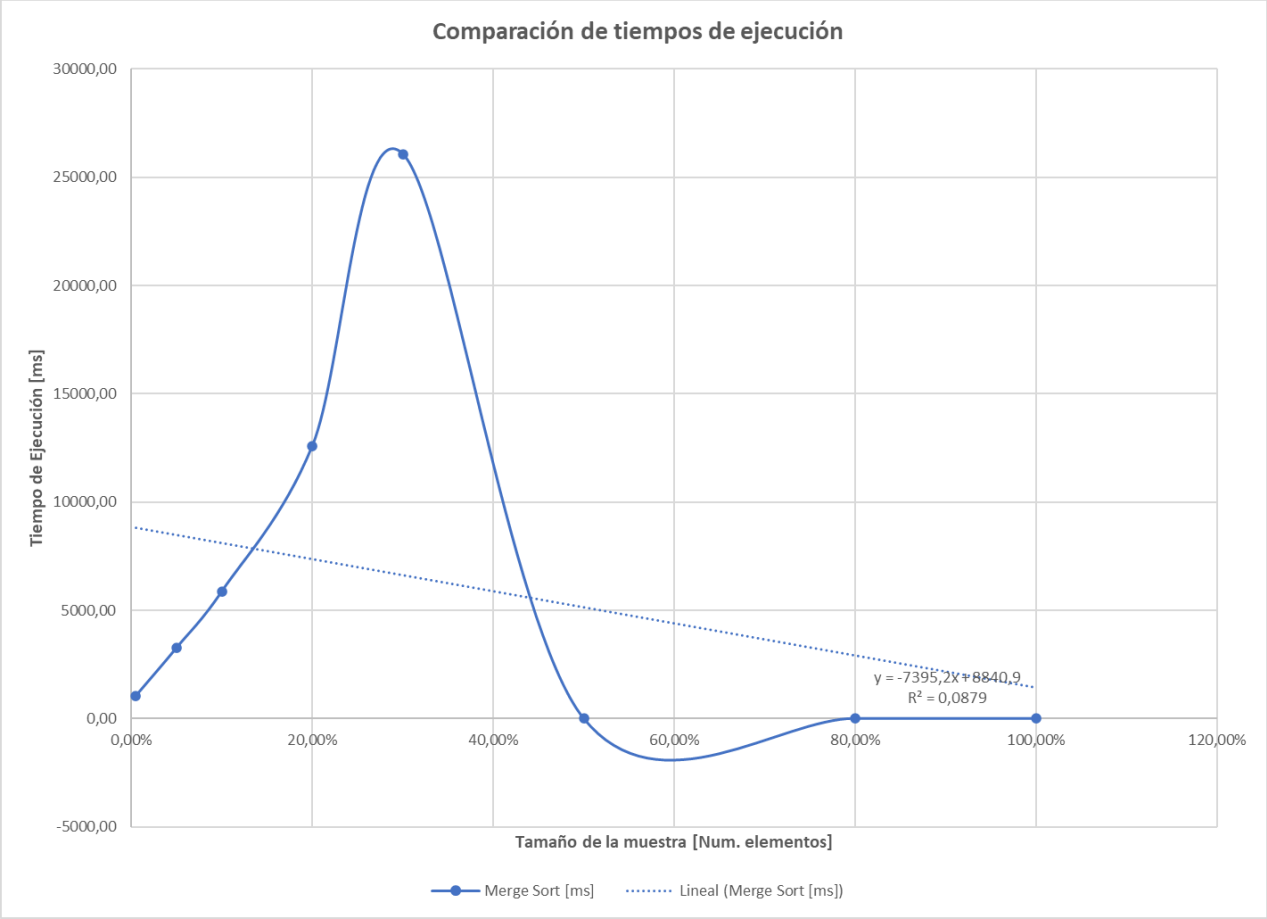
30 pct	26061.64
50 pct	—
80 pct	—
large	—

Tablas de datos

Muestra	Salida	Tiempo (ms)
small	Kosaraju, una lista con los componentes fuertemente conectados y la otra con el camino mas largo dentro de los componentes fuertemente conectados	1060.55
5 pct	Kosaraju, una lista con los componentes fuertemente conectados y la otra con el camino mas largo dentro de los componentes fuertemente conectados	3279.84
10 pct	Kosaraju, una lista con los componentes fuertemente conectados y la otra con el camino mas largo dentro de los componentes fuertemente conectados	5882.69
20 pct	Kosaraju, una lista con los componentes fuertemente conectados y la otra con el camino mas largo dentro de los componentes fuertemente conectados	12589.89
30 pct	Kosaraju, una lista con los componentes fuertemente conectados y la otra con el camino mas largo dentro de los componentes fuertemente conectados	26061.64
50 pct	Kosaraju, una lista con los componentes fuertemente conectados y la otra con el camino mas largo dentro de los componentes fuertemente conectados	—

80 pct	Kosaraju, una lista con los componentes fuertemente conectados y la otra con el camino mas largo dentro de los componentes fuertemente conectados	—
large	Kosaraju, una lista con los componentes fuertemente conectados y la otra con el camino mas largo dentro de los componentes fuertemente conectados	—

Graficas



Análisis

Como se puede ver de los tiempos de ejecución, la complejidad del requerimiento se vuelve más compleja a medida que aumenta tanto el número de vértices como el número de arcos que hay en la gráfica, y a pesar de también estar la complejidad de n log n por merge sort, esta es menos relevante a la de dfo, gracias a la mayor cantidad de arcos que vertices que se encuentran el la grafica