

# ANÁLISIS DEL RETO

*Franklin Smith Fernandez Romero, 202215104, F.fernandezr@uniandes.edu.co*

*Manuel Santiago Prieto Hernández, 202226947, m.prietoh@uniandes.edu.co*

*Pablo Arango Muriel, 202220340, p.arangom@uniandes.edu.co*

## Requerimiento 0 - Carga de datos

```
1 def load_moves(control, lista_eventos):
2     """
3     Función para agregar nuevos elementos a la lista
4     """
5     mapa = control["positions"]
6     grafo = control["moves"]
7
8     anterior = None
9
10    for evento in lt.iterator(lista_eventos):
11        punto = crear_identificador(evento)
12        individual_id = evento["individual-local-identifier"] + "_" + evento["tag-local-identifier"]
13        if not gr.containsVertex(grafo, punto):
14            gr.insertVertex(grafo, punto)
15            mp.put(mapa, punto, evento)
16
17        if anterior is not None and individual_id == anterior["individual-local-identifier"] + "_" + anterior["tag-local-identifier"]:
18            punto_ant = crear_identificador(anterior)
19            if gr.getEdge(grafo, punto_ant, punto) == None and (punto_ant != punto):
20                lon1 = round(float(anterior["location-long"]), 3)
21                lat1 = round(float(anterior["location-lat"]), 3)
22                lon2 = round(float(evento["location-long"]), 3)
23                lat2 = round(float(evento["location-lat"]), 3)
24
25                peso = haversine(lon1, lat1, lon2, lat2)
26                gr.addEdge(grafo, punto_ant, punto, peso)
27
28        anterior = evento
29
30    control["positions"] = mapa
31    control["moves"] = grafo
32
33    return control, gr.numVertices(grafo), gr.numEdges(grafo)
34
```

```

1 def agregar_encuentros(control):
2     grafo = control["moves"]
3     mapa = control["encuentros"]
4     mapa_positions = control["positions"]
5
6     lista = sort(mp.keySet(mapa_positions), 2)
7     anterior = None
8
9     for punto in lt.iterator(lista):
10        if anterior is not None:
11            iden1, lon1, lat1 = obtener_identificador_lon_lat(anterior)
12            iden2, lon2, lat2 = obtener_identificador_lon_lat(punto)
13            if (lon2, lat2) == (lon1, lat1):
14                encuentro = f"{lon1}_{lat1}"
15                if mp.contains(mapa, encuentro):
16                    gr.addEdge(grafo, encuentro, punto, 0)
17                    gr.addEdge(grafo, punto, encuentro, 0)
18                else:
19                    mp.put(mapa,encuentro, encuentro)
20                    mp.put(mapa_positions,encuentro, encuentro)
21                    gr.insertVertex(grafo, encuentro)
22                    gr.addEdge(grafo, encuentro, anterior, 0)
23                    gr.addEdge(grafo, anterior, encuentro, 0)
24                    gr.addEdge(grafo, encuentro, punto, 0)
25                    gr.addEdge(grafo, punto, encuentro, 0)
26            else:
27                encuentro = f"{lon2}_{lat2}"
28
29
30        else:
31            iden, lon2, lat2 = obtener_identificador_lon_lat(punto)
32
33            anterior = punto
34
35    control["moves"] = grafo
36    control["positions"] = mapa_positions
37    control["encuentros"] = mapa
38
39    return control, gr.numVertices(grafo), gr.numEdges(grafo), lista
40

```

## Descripción

En la carga de datos primero se obtiene todos los datos del csv de los lobos y los agrega en un mapa cuya llave son el identificador y el valor los datos del lobo, en el segundo csv, los eventos se guardan en un Array\_list, para luego ser ordenados por el identificador del lobo y la fecha de recorrido, a partir de este Array\_List se crea los puntos de seguimiento y los arcos de estos puntos en el grafo, además, se agregan a una tabla hash cuya llave es el punto de seguimiento y el valor es el evento. Luego se optiene una lista con los puntos de seguimiento del grafo, a partir de la tabla hash de positions, y se crea los puntos de encuentro a partir de ahí

<b>Entrada</b>	Control, FileNames del archivo
<b>Salidas</b>	Tablas hash de positions, wolfs y encuentros; y el grafo
<b>Implementado (Sí/No)</b>	Si

## Análisis de complejidad

Pasos	Complejidad
Crear Array List de los datos	$O(n)$
Crear mapa con los puntos de seguimiento	$O(n)$
Crear mapa con los puntos de encuentro	$O(n)$

Agregar puntos de encuentro y seguimiento	$O(n)$
Agregar arcos	$O(n)$
<b>Total</b>	$O(n)$

## Pruebas Realizadas

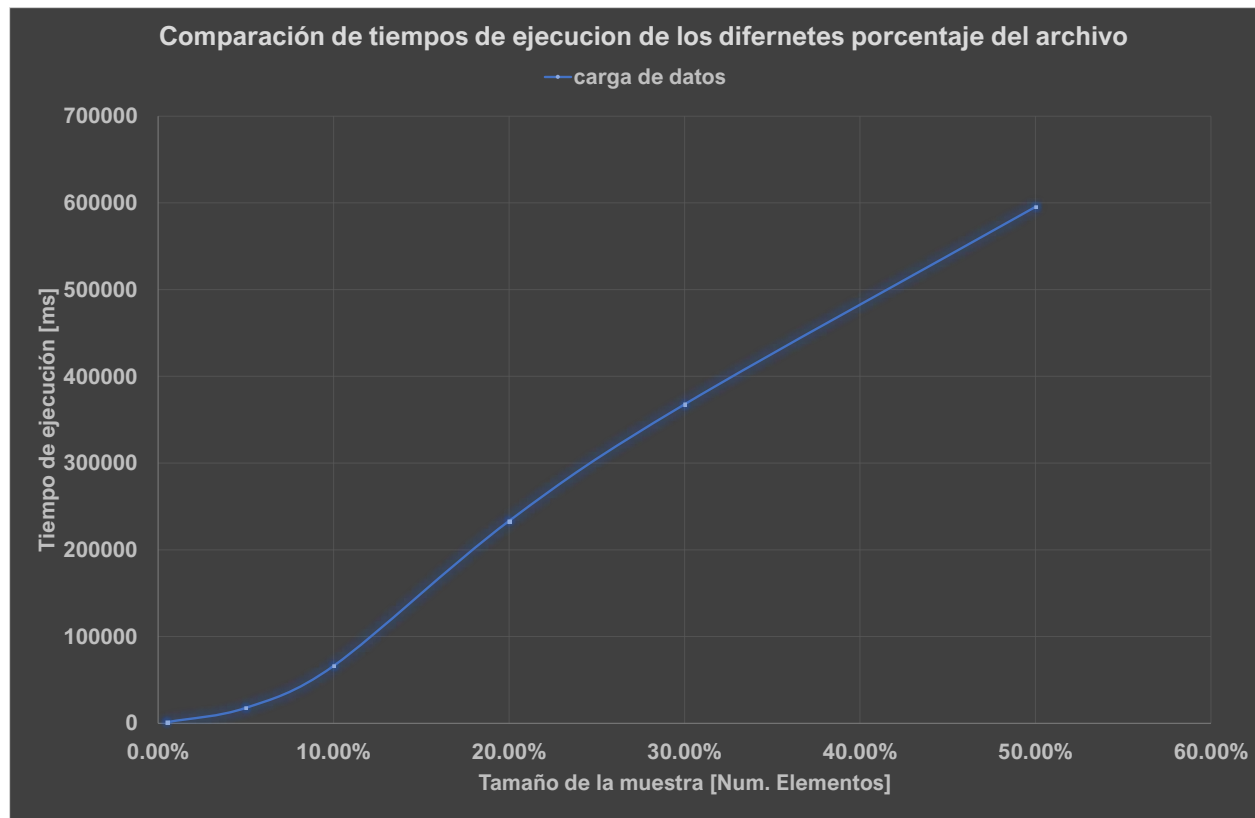
Las pruebas realizadas fueron realizadas en una maquina con las siguientes especificaciones. Los datos de entrada fueron el data structs del modelo.

Procesadores	<b>11th Gen Intel(R) Core(TM) i3-1115G4 @ 3.00GHz</b>
Memoria RAM	12 GB
Procesador	Core i3
Sistema Operativo	Windows 10

Entrada	Tiempo (s)
small	1181.523
5 pct	17771.54
10 pct	66126.61
20 pct	233266.2
30 pct	367744.3
50 pct	595491.6

## Graficas

Las gráficas con la representación de las pruebas realizadas.



## Análisis

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

Según el cálculo de complejidad se obtiene que la complejidad es  $O(n)$  y analizando la gráfica, podemos ver que se acerca bastante a la linealidad teniendo pequeñas variaciones debido a procesos corriendo en segundo plano en el computador.

En conclusión, si se ve una relación entre el cálculo teórico y el que se aprecia experimentalmente.

## Requerimiento 1

```

1 def req_1(data_structs, inc, fin):
2     """
3     Función que soluciona el requerimiento 1
4     """
5     # TODO: Realizar el requerimiento 1
6     recorrido= dfs.DepthFirstSearch(data_structs, inc)
7     puntos_en= 0
8     suma_arc=0
9     if dfs.hasPathTo(recorrido, fin):
10         path = dfs.pathTo(recorrido, fin)
11         size= st.size(path)
12         lista=lt.newList(datastructure="ARRAY_LIST")
13         while not st.isEmpty(path):
14             vertex = st.pop(path)
15             txt= vertex.split("_")
16             if len(txt)==2:
17                 puntos_en+=1
18                 data= crear_datos_req1(data_structs, vertex)
19                 if st.size(path) != size-1:
20                     if st.isEmpty(path):
21                         arco="Unknown"
22                         vertice= "Unknown"
23                         data= data, vertice, arco
24                     else:
25                         anterior= lt.lastElement(lista)
26                         arco= gr.getEdge(data_structs, anterior[2], data[2])
27                         suma_arc+= float(arco["weight"])
28                         anterior= anterior, vertex, arco["weight"]
29                         lt.removeLast(lista)
30                         lt.addLast(lista, anterior)
31                 lt.addLast(lista, data)
32         if req8_bool:
33             iterator = lt.iterator(lista)
34             ln_i = lt.firstElement(lista)[0][0]
35             lt_i = lt.firstElement(lista)[0][1]
36             m = folium.Map(location=[lt_i, ln_i], zoom_start=12)
37             trail = []
38             for i in iterator:
39                 if type(i[1]) != float:
40                     trail.append([i[0][1], i[0][0]])
41             folium.PolyLine(trail).add_to(m)
42             output_file = "req1.html"
43             m.save(output_file)
44             lista= cinco_prim_ult(lista)
45             return lista, size, puntos_en, suma_arc
46
47     else:
48         return False

```

```

1 def crear_datos_req1(grafo, vertex):
2     iden, lon, lat= obtener_identificador_lon_lat(vertex)
3     lon, lat= convertir_lon_lat(lon, lat)
4     individual_id= ""
5     if iden== 0:
6         lista= gr.adjacents(grafo, vertex)
7
8         for data in lt.iterator(lista):
9             iden2, lon2, lat2 = obtener_identificador_lon_lat(data)
10             individual_id= individual_id+","+ iden2
11             individual_id= individual_id.strip(",")
12     else:
13
14         individual_id= iden
15         individual_count= individual_id.split(",")
16         individual_count=len(individual_count)
17
18     return lon, lat, vertex, individual_id, individual_count
19

```

Descripción

Este requerimiento obtiene un camino entre dos puntos de encuentro. Para esto recibe un punto de origen y un punto de fin, y utiliza dfs para calcular la distancia, los puntos y los arcos.

<b>Entrada</b>	Grafo, punto inc y fin
<b>Salidas</b>	Lista vertices, num puntos de encuentro, puntos recorrido, suma arcos
<b>Implementado (Sí/No)</b>	Si

## Análisis de complejidad

Pasos	Complejidad
Aplicar dfs	$O(V+E)$
Obtener pila del recorrido hasta el punto final	$O(V+E)$
Desencolar pila	$O(V)$
Obtener arcos	$O(V)$
<b>Total</b>	$O(V+E)$

## Pruebas Realizadas

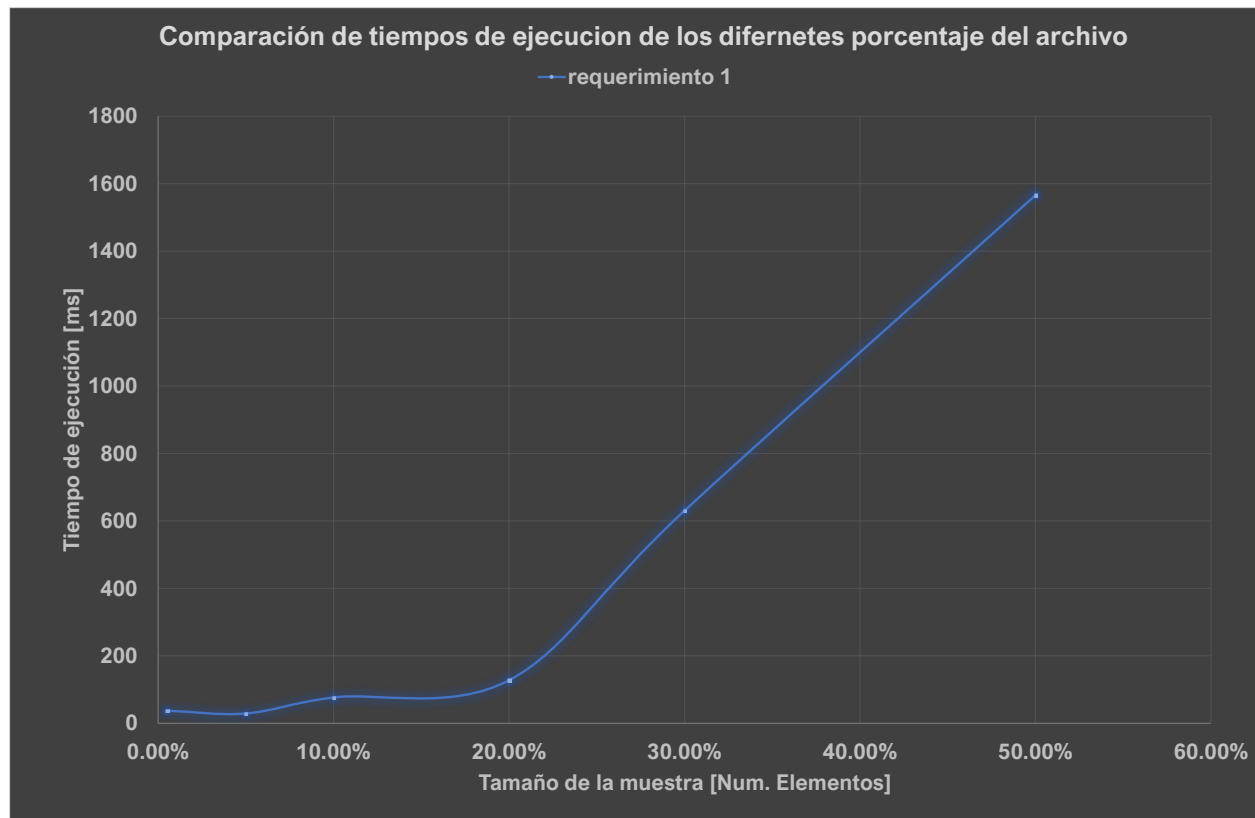
Las pruebas realizadas fueron realizadas en una maquina con las siguientes especificaciones. Los datos de entrada fueron el data structs del modelo.

Procesadores	<b>11th Gen Intel(R) Core(TM) i3-1115G4 @ 3.00GHz</b>
Memoria RAM	12 GB
Procesador	Core i3
Sistema Operativo	Windows 10

Entrada	Tiempo (s)
small	37.449
5 pct	29.157
10 pct	76.7
20 pct	127.722
30 pct	630.433
50 pct	1565.6521

## Graficas

Las gráficas con la representación de las pruebas realizadas.



## Análisis

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

Según el cálculo de complejidad se obtiene que la complejidad es  $O(n)$  y analizando la gráfica, podemos ver que se acerca bastante a la linealidad teniendo pequeñas variaciones debido a procesos corriendo en segundo plano en el computador.

En conclusión, si se ve una relación entre el cálculo teórico y el que se aprecia experimentalmente.

## Requerimiento 2

```

1 def req_2(data_structs, inc, fin):
2     """
3     Función que soluciona el requerimiento 2
4     """
5     # TODO: Realizar el requerimiento 2
6     recorrido= bfs.BreadthFirstSearch(data_structs, inc)
7     puntos_en= 0
8     suma_arc=0
9     if bfs.hasPathTo(recorrido, fin):
10         path = bfs.pathTo(recorrido, fin)
11         size= st.size(path)
12         lista=lt.newList(datastructure="ARRAY_LIST")
13         while not st.isEmpty(path):
14             vertex = st.pop(path)
15             txt= vertex.split("_")
16             if len(txt)==2:
17                 puntos_en+=1
18                 data= crear_datos_req1(data_structs, vertex)
19                 if st.size(path) != size-1:
20                     if st.isEmpty(path):
21                         arco="Unknown"
22                         vertice= "Unknown"
23                         data= data, vertice, arco
24                     else:
25                         anterior= lt.lastElement(lista)
26                         arco= gr.getEdge(data_structs, anterior[2], data[2])
27                         suma_arc+= float(arco["weight"])
28                         anterior= anterior, vertex, arco["weight"]
29                         lt.removeLast(lista)
30                         lt.addLast(lista, anterior)
31                 lt.addLast(lista, data)
32             if req8_bool:
33                 iterator = lt.iterator(lista)
34                 ln_i = lt.firstElement(lista)[0][0]
35                 lt_i = lt.firstElement(lista)[0][1]
36                 m = folium.Map(location=[lt_i, ln_i], zoom_start=12)
37                 trail = []
38                 for i in iterator:
39                     if type(i[1]) != float:
40                         trail.append([i[0][1], i[0][0]])
41                 folium.PolyLine(trail).add_to(m)
42                 output_file = "req2.html"
43                 m.save(output_file)
44                 return lista, size, puntos_en, suma_arc
45             else:
46                 return False
47

```

## Descripción

Este requerimiento obtiene un camino con el menor número de puntos, entre dos puntos de encuentro. Para esto recibe un punto de origen y un punto de fin, y utiliza bfs para calcular la distancia, los puntos y los arcos.

<b>Entrada</b>	Grafo, punto inc y fin
<b>Salidas</b>	Lista vértices, núm. puntos de encuentro, puntos recorridos, suma arcos
<b>Implementado (Sí/No)</b>	Sí

## Análisis de complejidad

Pasos	Complejidad
Aplicar bfs	$O(V+E)$
Obtener pila del recorrido hasta el punto final	$O(V+E)$
Desencolar pila	$O(V)$
Obtener arcos	$O(V)$



<b>Total</b>	<b>O(V+E)</b>
--------------	---------------

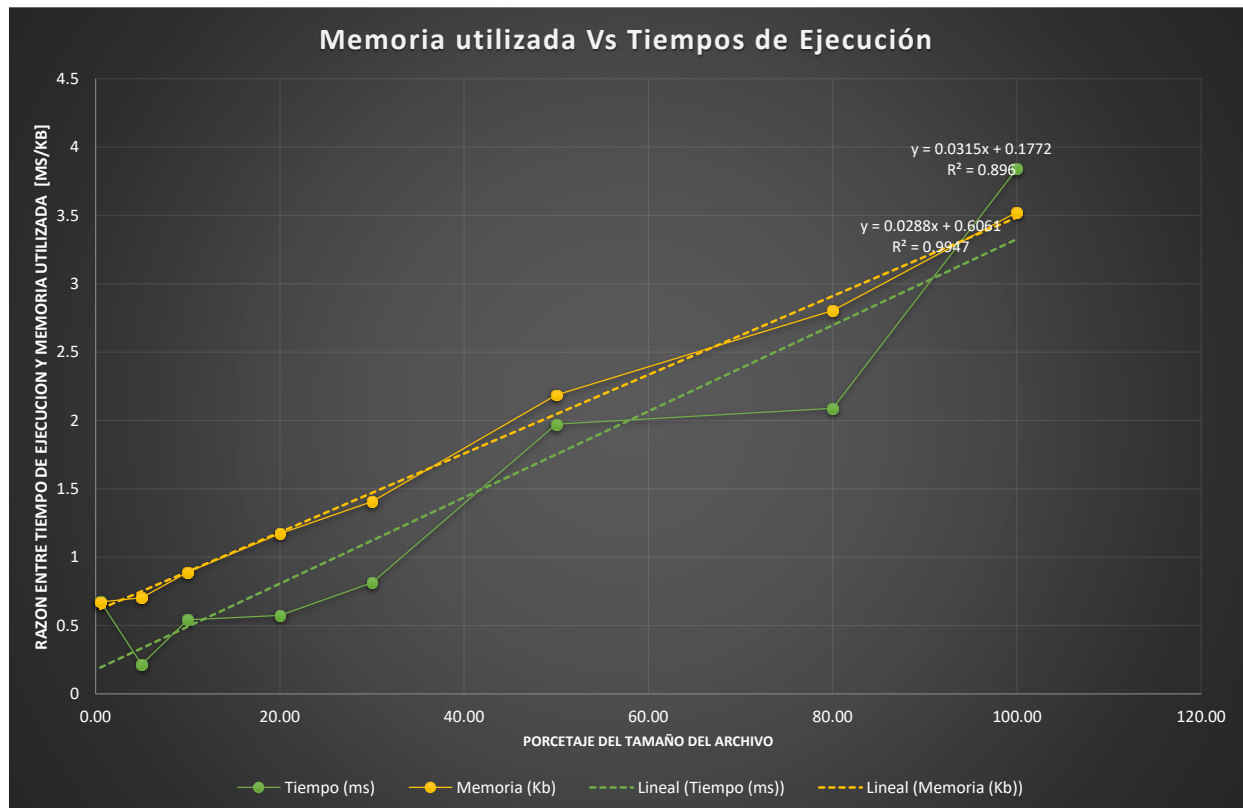
## Pruebas Realizadas

<b>Procesadores</b>	11th Gen Intel(R) Core(TM) i3-1115G4 @ 3.00GHz
<b>Memoria RAM</b>	12 GB
<b>Procesador</b>	Core i3
<b>Sistema Operativo</b>	Windows 10

<b>Entrada</b>	<b>Tiempo (s)</b>
small	37.449
5 pct	29.157
10 pct	76.7
20 pct	127.722
30 pct	630.433
50 pct	1565.6521

## Graficas

Las gráficas con la representación de las pruebas realizadas.



## Análisis

Según el cálculo de complejidad se obtiene que la complejidad es  $O(n)$  y analizando la gráfica, podemos ver que se acerca bastante a la linealidad teniendo pequeñas variaciones debido a procesos corriendo en segundo plano en el computador.

En conclusión, se ve una relación entre el cálculo teórico y el que se aprecia experimentalmente.

## Requerimiento 3

```

1  def req_3(data_structs, clase_acc, Direccion):
2      """
3      Función que soluciona el requerimiento 3
4      """
5      # TODO: Realizar el requerimiento 3
6      arbol_key_val= mp.get(data_structs, "casos")
7      arbol_array= me.getValue(arbol_key_val)
8      arbol_hash= om.get(arbol_array, clase_acc)
9      registro= me.getValue(arbol_hash)
10
11
12      respuesta= om.newMap("RBT",
13                          compare_arbol_caso)
14
15      for data in lt.iterator(registro):
16          Direccion_data= data["DIRECCION"]
17          if Direccion.lower() in Direccion_data.lower():
18              om.put(respuesta, data["FORMULARIO"], data)
19
20      return sort(om.valueSet(respuesta), 5)

```

## Descripción

usa el algoritmo de Kosaraju para obtener los componentes fuertemente conectados y crea un mapa con los datos, donde el SCCID es la llave y los valores son una lista con los puntos

<b>Entrada</b>	árbol de casos, tipo de caso, y la calle específica
<b>Salidas</b>	Una Array List con todos los datos de esa calle y tipo de daño
<b>Implementado (Sí/No)</b>	Sí, implementado por Franklin Romero

## Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	Complejidad
Traer casos	$O(\log n)$
Traer caso específico	$O(\log n)$
Recorrido de cada lista de cada registro	$O(n)$
Comparar direcciones "calles o vías etc"	$O(n)$
Agregar a la lista	$O(n)$
Ordenar por MergeSort	$O(N \log n)$
TOTAL	$O(N \log n)$

## Pruebas Realizadas

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

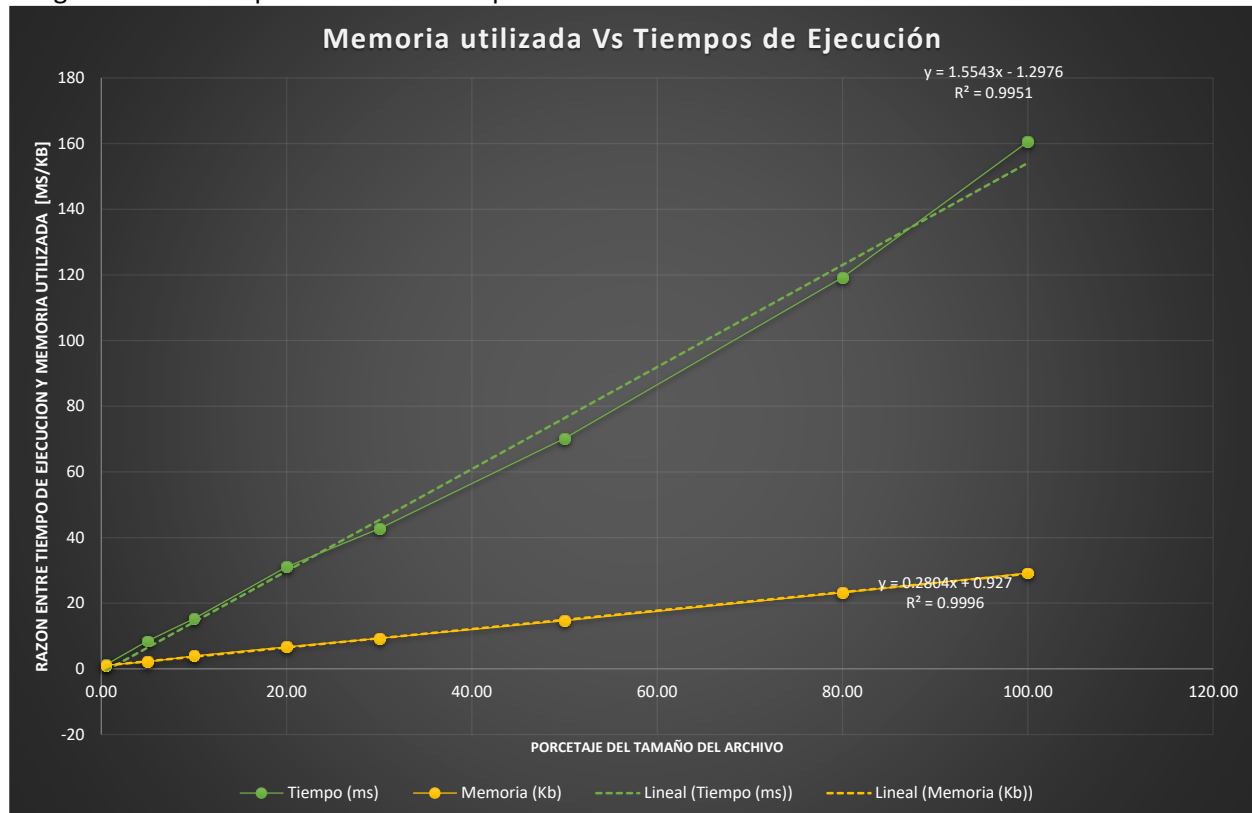
Procesadores	11th Gen Intel(R) Core(TM) i3-1115G4 @ 3.00GHz
Memoria RAM	12 GB
Procesador	Core i3

Sistema Operativo	Windows 10
-------------------	------------

Entrada	Tiempo (s)
small	37.449
5 pct	29.157
10 pct	76.7
20 pct	127.722
30 pct	630.433
50 pct	1565.6521

## Graficas

Las gráficas con la representación de las pruebas realizadas.



## Análisis

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

Aunque la complejidad calculada fue de  $O(N \log n)$ , en la gráfica se puede evidenciar que el código tiende a ser este mismo, por tal motivo se cree que el código en los mejores casos puede ser  $O(N \log n)$

## Requerimiento 4

```

def req_4(data, ori_lon, ori_lat, des_lon, des_lat):
    """
    Función que soluciona el requerimiento 4
    """
    # TODO: Realizar el requerimiento 4
    cont = 1
    lista = lt.newList(datastructure="ARRAY_LIST")
    lista2 = lt.newList(datastructure="ARRAY_LIST")
    lobos = lt.newList(datastructure="ARRAY_LIST")
    lista_mapa = lt.newList(datastructure="ARRAY_LIST")
    lista_enc = lt.newList(datastructure="ARRAY_LIST")
    mapa_positions = data["positions"]
    positions = data["encuentros"]
    lista_positions = mp.keySet(positions)
    p1 = lt.firstElement(lista_positions)
    _, dist1_lon, dist1_lat = obtener_identificador_lon_lat(p1)
    dist1_lon, dist1_lat = convertir_lon_lat(dist1_lon, dist1_lat)
    dist_ori = haversine(dist1_lon, dist1_lat, ori_lon, ori_lat)
    dist_des = haversine(dist1_lon, dist1_lat, des_lon, des_lat)
    ori = p1
    des = p1
    for pos in lt.iterator(lista_positions):
        _, lon, lat = obtener_identificador_lon_lat(pos)
        lon, lat = convertir_lon_lat(lon, lat)
        dist_ori_pos = haversine(lon, lat, ori_lon, ori_lat)
        dist_des_pos = haversine(lon, lat, des_lon, des_lat)
        if dist_ori_pos < dist_ori:
            dist_ori = dist_ori_pos
            ori = pos
        if dist_des_pos < dist_des:
            dist_des = dist_des_pos
            des = pos
    grafo = data["moves"]
    rec = djik.Dijkstra(grafo, ori)
    if djik.hasPathTo(rec, des):
        costo = djik.distTo(rec, des)
        camino = djik.pathTo(rec, des)
    if djik.hasPathTo(rec, des):
        costo = djik.distTo(rec, des)
        camino = djik.pathTo(rec, des)
        num_nodos = st.size(camino)
        for _ in range(num_nodos):
            ele = st.pop(camino)
            vertice = ele["vertexA"]
            lt.addLast(lista, vertice)
        vertice = ele["vertexB"]
        lt.addLast(lista, vertice)
        for i in lt.iterator(lista):
            id, a, b = obtener_identificador_lon_lat(i)
            if id == 0:
                dato = crear_datos_req4(grafo, i)
                lt.addLast(lista_enc, i)
            else:
                if not lt.isPresent(lobos, str(id)):
                    lt.addLast(lobos, str(id))
                    dato = crear_datos_req4(grafo, i)
                    lt.addLast(lista_mapa, dato)
        num_arcos = num_nodos
        for c in lt.iterator(lista_enc):
            cont += 1
            dato = crear_datos_req4(grafo, c)
            if c != lt.lastElement(lista):
                dist_nxt = djik.distTo(rec, lt.getElement(lista_enc, cont)) - djik.distTo(rec, c)
            else:
                dist_nxt = 0
            dato.append(dist_nxt)
            lt.addLast(lista2, dato)

```

La función req\_4 resuelve el requerimiento de identificar el corredor migratorio entre dos puntos en una región. La función inicializa variables y estructuras de datos, calcula la distancia entre el punto de origen y aquel dado y hace lo mismo con el de destino. Después, a partir del grafo cargado, utiliza el algoritmo de Dijkstra para encontrar el camino más corto, guarda los datos de cada nodo del camino y construye listas con información de los nodos. Si no hay ningún camino asigna valores "Desconocido" a la mayoría de los valores. Para terminar, obtiene una lista final y el tamaño de la lista de nodos, y finalmente devuelve un conjunto de datos con la distancia de origen, distancia de destino, costo, tamaño de listas y número de arcos.

## Descripción

Esta función retorna los 5 accidentes más recientes en un periodo de tiempo.

<b>Entrada</b>	La estructura de datos principal, longitud inicial, latitud inicial, longitud final, latitud final
<b>Salidas</b>	Distancia origen, distancia destino, costo, lista con datos de cada punto, numero de arcos.
<b>Implementado (Sí/No)</b>	Si, implementado por Pablo Arango

## Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

<b>Pasos</b>	<b>Complejidad</b>
Crear una lista vacía	$O(1)$
Buscar nodo más cercano	$O(n)$
Crear recorrido Dijkstra	$O((v+e) \log v)$
Obtener camino de menor costo a partir de Dijkstra	$O(n)$
Crear lista final	$O(n)$
<b>TOTAL</b>	<b><math>O((v+e) \log v)</math></b>

## Pruebas Realizadas

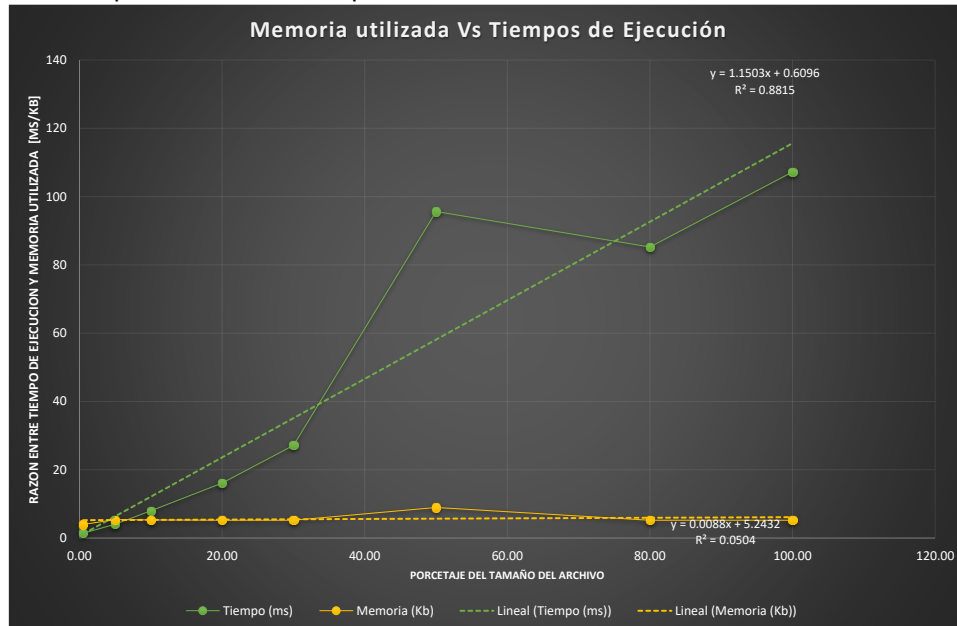
Las pruebas realizadas fueron realizadas en una maquina con las siguientes especificaciones. Los datos de entrada fueron el data structs del modelo.

Procesadores	<b>11th Gen Intel(R) Core(TM) i3-1115G4 @ 3.00GHz</b>
Memoria RAM	12 GB
Procesador	Core i3
Sistema Operativo	Windows 10

<b>Entrada</b>	<b>Tiempo (s)</b>
small	37.449
5 pct	29.157
10 pct	76.7
20 pct	127.722
30 pct	630.433
50 pct	1565.6521

## Graficas

Las gráficas con la representación de las pruebas realizadas.



## Análisis

Aunque la complejidad calculada fue de  $O((v+e) \log v)$ , en la gráfica se puede evidenciar que el código tiende a ser lineal, por tal motivo se cree que el código en los mejores casos puede ser  $O(v+e)$ . Es evidente que el  $\log(v)$  no tiene este impacto en el caso promedio, pues en este caso solo deberá tomar en cuenta los datos del recorrido específico.

## Requerimiento 5

```

1 def req_5(data_structs, puntos, kil, inc):
2     """
3     Función que soluciona el requerimiento 5
4     """
5     # TODO: Realizar el requerimiento 5
6     grafo= data_structs["moves"]
7     mapa_positions= data_structs["positions"]
8     lista_positions= mp.keySet(mapa_positions)
9     kil= (float(kil)/2)
10    recorridos= bf.BellmanFord(grafo, inc)
11    encuentros=om.newMap("BST",
12                        compare_arbol_caso)
13    for encuentro in lt.iterator(lista_positions):
14
15        costo=bf.distTo(recorridos, encuentro)
16        if costo<= kil:
17            om.put(encuentros, costo, encuentro)
18
19    rutas= om.size(encuentros)
20    if rutas!=0:
21        valor= obtener_recorrido_max(recorridos, encuentros, puntos)
22        if valor!= False:
23            recorrido mayor, distancia, min_pun= valor
24            lista mapa = lt.newList(datastructure="ARRAY_LIST")
25            lista_vertices=lt.newList(datastructure="ARRAY_LIST")
26            lista_animales=lt.newList(datastructure="ARRAY_LIST")
27            size= st.size(recorrido mayor)
28            a = True
29            while not st.isEmpty(recorrido mayor):
30                vertex_tot = st.pop(recorrido mayor)
31                vertex= vertex_tot["vertexA"]
32                lt.addLast(lista_vertices, vertex)
33                if a:
34                    lt.addLast(lista_mapa, vertex_tot["vertexB"])
35                    a = not a
36                lt.addLast(lista_mapa, vertex)
37                lista_vertices= sort(lista_vertices, 2)
38                lista_ver_2= lista_vertices
39                for vertex in lt.iterator(lista_ver_2):
40                    txt= vertex.split(".")
41                    if len(txt)==2:
42                        animals= gr.outdegree(grafo, vertex)
43                    else:
44                        animals=1
45                lt.addLast(lista_animales, animals)
46            respuesta= size, distancia, lista_vertices, lista_animales
47            if req_8_bool:
48                iterator= lt.iterator(lista_mapa)
49                ln_i, lt_i = obtener_identificador_lon_lat(lt.firstElement(lista_mapa))
50                ln_i, lt_i = convertir_lon_lat(ln_i, lt_i)
51                m = folium.Map(location=[lt_i, ln_i], zoom_start=10)
52                trail = []
53                for i in iterator:
54                    l_ln, l_lt = obtener_identificador_lon_lat(i)
55                    l_ln, l_lt = convertir_lon_lat(l_ln, l_lt)
56                    trail.append([l_ln, l_ln])
57                folium.PolyLine(trail).add_to(m)
58                output_file = "req5.html"
59                m.save(output_file)
60                return rutas, min_pun, distancia*2, respuesta
61
62    else:
63        return False
64
65    return False

```

```

1 def obtener_recorrido_max(recorridos, mapa, valor):
2     lista= om.keySet(mapa)
3     while lt.size(lista) != 0:
4         distancia_max= om.maxKey(mapa)
5         entry= om.get(mapa, distancia_max)
6         value= me.getValue(entry)
7         path= bf.pathTo(recorridos, value)
8         puntos= st.size(path)
9         if puntos>= int(valor):
10             return path, distancia_max, puntos
11         else:
12             om.deleteMax(mapa)
13             lt.removeLast(lista)
14     return False

```



## Descripción

Esta función obtiene el corredor migratorio más extenso que puedo revisar desde un punto de encuentro específico. Para esto empieza por hacer un recorrido en Bellman Ford desde el punto inicial, luego va comparando los costos de llegada en cada punto de encuentro y los filtra para obtener los menores a los kilómetros posibles del guardabosques, y los guarda en un árbol. Luego obtiene el mayor y va comparando si cuenta con los puntos de encuentros mínimos, sino lo borra del árbol y busca el siguiente. Luego retorna el número de rutas posibles, la mejor ruta y la distancia que tien que recorrer en esa ruta el guardabosques.

<b>Entrada</b>	Grafo, min puntos de encuentro, los kilómetros y el punto inicial
<b>Salidas</b>	Num rutas, distancia, lista datos del recorrido
<b>Implementado (Sí/No)</b>	Si, implementado por Manuel Prieto

## Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

<b>Pasos</b>	<b>Complejidad</b>
Aplicar Belman Ford	$O((v+e) \log v)$
Obtener la distancia de los puntos de encuentro	$O(n(v \log e))$
Agregar al mapa	$O(v)$
Recorrer el mapa	$O(v)$
Obtener puntos	$O(v)$
<b>TOTAL</b>	<b><math>O(n(v \log e))</math></b>

## Pruebas Realizadas

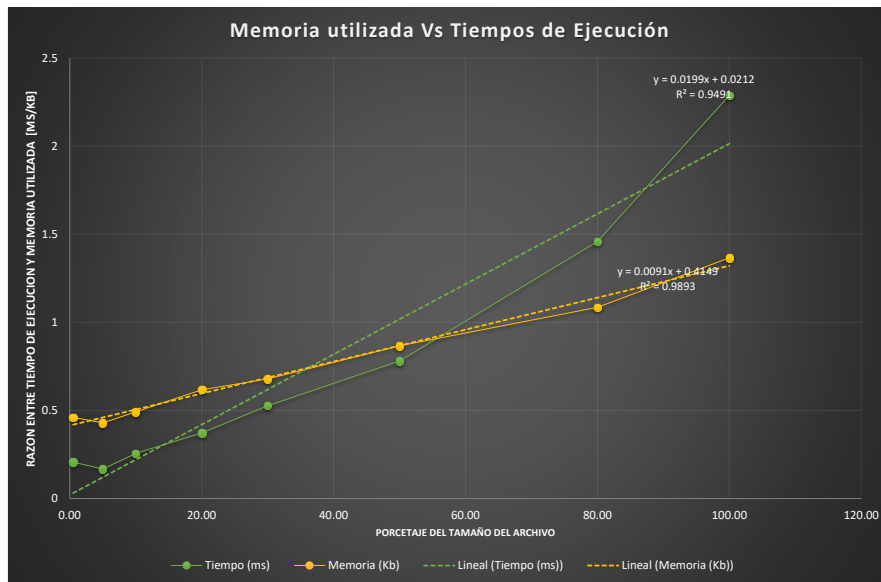
Las pruebas realizadas fueron realizadas en una maquina con las siguientes especificaciones. Los datos de entrada fueron el data structs del modelo.

Procesadores	11th Gen Intel(R) Core(TM) i3-1115G4 @ 3.00GHz
Memoria RAM	12 GB
Procesador	Core i3
Sistema Operativo	Windows 10

<b>Entrada</b>	<b>Tiempo (s)</b>
small	37.449
5 pct	29.157
10 pct	76.7
20 pct	127.722
30 pct	630.433
50 pct	1565.6521

## Graficas

Las gráficas con la representación de las pruebas realizadas.



## Análisis

Aunque la complejidad calculada fue de  $O(n)$ , en la gráfica se puede evidenciar que el código tiende a ser lineal, por tal motivo se cree que el código en los mejores casos puede ser  $O(\log n)$ .

## Requerimiento 6

```

1 def res_G(control, inc, fan, pen):
2     """
3     función que actualiza el requerimiento G
4     """
5     # Se crea un grafo y se crea una lista de eventos en el rango temporal, y se define por lado y fecha
6     arbol = crear_arbol_fecha(control["positions"])
7     lista_rango = lista_eventos_arbol(arbol, inc, fan)
8     lista_rango = sort(lista_rango, 1)
9
10    # Se crea una lista de eventos y se crea una lista de valores
11    wolf = np.zeros(len(control["wolves"]))
12    wolf = gen_it_newlist(datastructure="ARBOY_LIST")
13    for wolf in lt.iterator(wolf):
14        if wolf["animal-inc"] != 0:
15            it.addset(wolf["gen"], wolf)
16
17    # Se crea un mapa con listas de eventos de cada lado
18
19    mapa_wolf = crear_lista_con_lados_eventos(wolf["gen"], lista_rango)
20
21    # Se crea los mapas y se crean los mapas a partir del mapa y el grafo
22    mayor = -999999
23    menor = 999999
24    lista_mayores = []
25    lista_menores = []
26    lista_wolf = np.zeros(len(mapa_wolf))
27    lista_may = ""
28    lista_men = ""
29
30    grafo = gr.newGraph(datastructure="AD_LIST",
31                        directed=False,
32                        size=10000,
33                        repFunction=compareStrings)
34
35    for eventos in lt.iterator(lista_wolf):
36        grafo.agregar_al_grafo(grafo, eventos)
37        # Se calcula la distancia total del grafo, eventos
38        if peso > mayor:
39            mayor = peso
40        # Se calcula la distancia total del grafo, eventos
41        lista_may = eventos
42        iden_may = lt.getFirstElement(eventos)
43        iden_may = iden_may["individual-local-identifier"] + "-" + iden_may["lac-local-identifier"]
44        # Se calcula la distancia total del grafo, eventos
45        lista_men = eventos
46        iden_men = lt.getFirstElement(eventos)
47        iden_men = iden_men["individual-local-identifier"] + "-" + iden_men["lac-local-identifier"]
48
49    wolf_may = np.get(control["wolves"], iden_may)
50    wolf_men = np.get(control["wolves"], iden_men)
51    nodos_may = lt.iterator(lista_may)
52    nodos_men = lt.iterator(lista_men)
53    if res_G_hol:
54        iterator = lt.iterator(lista_may)
55        i = 0
56        while i < len(iterator):
57            i += 1
58            i = 1
59            i = 1
60            i = 1
61            i = 1
62            i = 1
63            i = 1
64            i = 1
65            i = 1
66            i = 1
67            i = 1
68            i = 1
69            i = 1
70            i = 1
71            i = 1
72            i = 1
73            i = 1
74            i = 1
75            i = 1
76            i = 1
77            i = 1
78            i = 1
79            i = 1
80            i = 1
81            i = 1
82            i = 1
83            i = 1
84            i = 1
85            i = 1
86            i = 1
87            i = 1
88            i = 1
89            i = 1
90            i = 1
91            i = 1
92            i = 1
93            i = 1
94            i = 1
95            i = 1
96            i = 1
97            i = 1
98            i = 1
99            i = 1
100           i = 1

```

```

1 def agregar_al_grafo(grafo, lista):
2     anterior=None
3     for data in lt.iterator(lista):
4         punto= crear_identificador(data)
5         gr.insertVertex(grafo, punto)
6         if anterior is not None:
7             punto_ant= crear_identificador(anterior)
8             lon1 = round(float(anterior["location-long"]), 3)
9             lat1 = round(float(anterior["location-lat"]), 3)
10            lon2 = round(float(data["location-long"]), 3)
11            lat2 = round(float(data["location-lat"]), 3)
12
13            peso = haversine(lon1, lat1, lon2, lat2)
14            gr.addEdge(grafo, punto_ant, punto, peso)
15
16            anterior= data
17
18    return grafo


```



```
1 def obtener_distancia_total(grafo, lista):
2     peso=0
3     first= lt.firstElement(lista)
4     punto= crear_identificador(first)
5     last= lt.lastElement(lista)
6     punto_last= crear_identificador(last)
7     search= djik.Dijkstra(grafo, punto)
8     peso= djik.distTo(search, punto_last)
9
10    return peso
```




```
1 def crear_tabla_con_lobos_eventos(wolfs_gen, lista_rango):
2     mapa_event= mp.newMap(200,
3                             maptype='PROBING',
4                             loadfactor=0.5,
5                             cmpfunction=compare_map)
6     lista_eventos= lt.newList(datastructure="ARRAY_LIST")
7     first= lt.firstElement(lista_rango)
8     anterior= None
9     for evento in lt.iterator(lista_rango):
10        individual_id=evento["individual-local-identifier"]+"_"+evento["tag-local-identifier"]
11        if individual_id == anterior:
12            entry= mp.get(mapa_event, individual_id)
13            value= me.getValue(entry)
14            lt.addLast(value, evento)
15            mp.put(mapa_event, individual_id, value)
16        else:
17            lista_eventos= lt.newList(datastructure="ARRAY_LIST")
18            lt.addLast(lista_eventos, evento)
19            mp.put(mapa_event, individual_id, lista_eventos)
20
21        anterior= individual_id
22
23    mapa_wolfs= mp.newMap(200,
24                          maptype='PROBING',
25                          loadfactor=0.5,
26                          cmpfunction=compare_map)
27
28    for wolf in lt.iterator(wolfs_gen):
29        individual_id=wolf["animal-id"]+"_"+wolf["tag-id"]
30        entry= mp.get(mapa_event, individual_id)
31        if entry != None:
32            value= me.getValue(entry)
33            mp.put(mapa_wolfs, individual_id, value)
34
35    return mapa_wolfs
```



```

1 def lista_eventos_rango(arbol, inc, fin):
2     lista_rango= om.values(arbol, inc, fin)
3     lista_eventos=lt.newList(datastructure="ARRAY_LIST")
4     for data in lt.iterator(lista_rango):
5         for evento in lt.iterator(data):
6             lt.addLast(lista_eventos, evento)
7
8     return lista_eventos

```



```

1 def crear_arbol_fechas(mapa):
2     lista= crear_lista_movimientos(mapa)
3     lista= sort(lista, 4)
4     arbol=om.newMap("RBT",
5                   compare_arbol_caso)
6
7     lista_valores= lt.newList(datastructure="ARRAY_LIST")
8     first= lt.firstElement(lista)
9     fecha=first["timestamp"]
10    for data in lt.iterator(lista):
11        fecha_data=data["timestamp"]
12        if data== first:
13            lt.addLast(lista_valores, data)
14            om.put(arbol, fecha_data, lista_valores)
15        elif fecha_data== fecha:
16            entry= om.get(arbol, fecha_data)
17            value= me.getValue(entry)
18            lt.addLast(value, data)
19            om.put(arbol, fecha, value)
20        else:
21            lista_valores= lt.newList(datastructure="ARRAY_LIST")
22            lt.addLast(lista_valores, data)
23            om.put(arbol, fecha_data, lista_valores)
24            fecha= fecha_data
25
26    return arbol

```

```

1  def crear_lista_movimientos(mapa):
2      lista=lt.newList(datastructure="ARRAY_LIST")
3      lista_posiciones= mp.valueSet(mapa)
4      for data in lt.iterator(lista_posiciones):
5          if type(data) != str:
6              lt.addLast(lista, data)
7
8      return lista

```

## Descripción

Esta función obtiene el comportamiento de los lobos del estudio según el sexo registrado. Para esto primero crea un mapa con los eventos, dividiéndolos por fecha, luego obtiene los datos de ese rango de fechas. Divide los lobos por género, y luego con los valores del rango se buscan los lobos en común. Al final se crea el grafo a partir de esto y se aplica DJK.

<b>Entrada</b>	Control, fecha inicial, fecha final y el genero
<b>Salidas</b>	mayor, iden_may, wolf_may, lista_may, nodos_may, menor, iden_men, wolf_men, lista_men, nodos_men, gr.numEdges(grafo), gr.numVertices(grafo)
<b>Implementado (Sí/No)</b>	Sí

## Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	Complejidad
Crear arbol eventos	$O(n)$
Obtener rango	$O(\log n)$
Filtrar lobos	$O(n)$
Juntar datos	$O(n)$
Crear grafo	$O((v+e)\log v)$
Aplicar DJK	$O((v+e)\log v)$
<b>TOTAL</b>	$O((v+e)\log v)$

## Pruebas Realizadas

Las pruebas realizadas fueron realizadas en una maquina con las siguientes especificaciones. Los datos de entrada fueron el data structs del modelo.

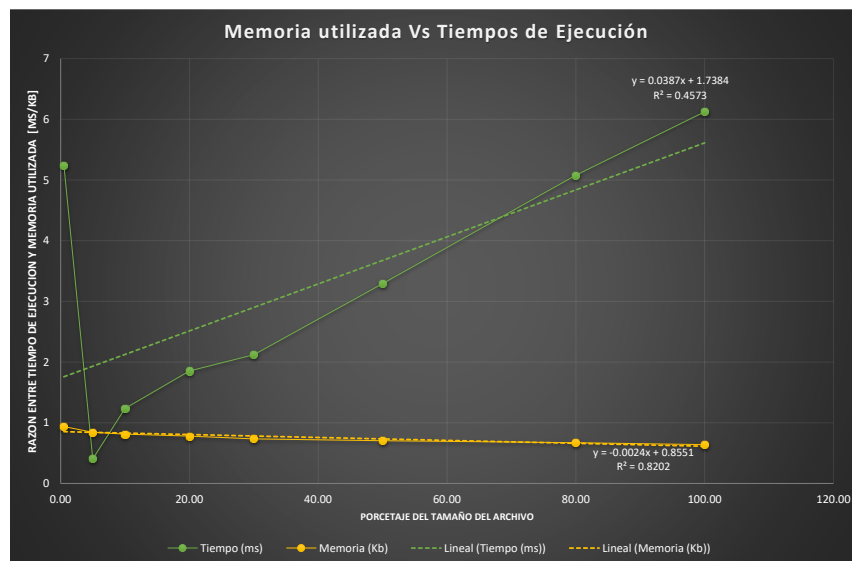
Procesadores	11th Gen Intel(R) Core(TM) i3-1115G4 @ 3.00GHz
Memoria RAM	12 GB
Procesador	Core i3

Sistema Operativo	Windows 10
-------------------	------------

Entrada	Tiempo (s)
small	37.449
5 pct	29.157
10 pct	76.7
20 pct	127.722
30 pct	630.433
50 pct	1565.6521

## Graficas

Las gráficas con la representación de las pruebas realizadas.



## Análisis

La gráfica se comporta según lo calculado en la complejidad del código, ya que podemos ver que, con respecto a los tiempos, la gráfica tiende a ser  $n \log n$ , y lo calculado en la complejidad es  $O(\log n)$

## Requerimiento 7

```
1 def req_(control, inc, fin, tem_min, tem_max):
2     """
3     función que analiza el requerimiento /
4     """
5     #crea un arbol y una lista con los eventos en el rango ingresado, y lo ordena por fecha y fecha
6     arbol= crear_arbol_fecha(control["positions"])
7     lista_rango= lista_eventos_rango(arbol, inc, fin)
8     lista_rango= sort(lista_rango, 1)
9
10    #Crea lista de eventos aparte de la temperatura
11    lista_eventos=ll.newList(datastructure="ARRAY LIST")
12    for evento in lt.iterator(lista_rango):
13        if float(evento["external-temperature"])> float(tem_min) and float(evento["external-temperature"])<float(tem_max):
14            lt.addlast(lista_eventos, evento)
15
16    lista_eventos= sort(lista_eventos, 1)
17    grafo= crear_grafo_manadas(lista_eventos)
18
19    #se ejecuta Kosaraju y se organizan los datos
20    search= cc.KosarajuBFS(grafo)
21    mapa_idsc= search["idsc"] #mapa que tiene como llaves los nodos del grafo, y como valor el scid
22    keys= mp.keySet(mapa_idsc)
23    #mapa que tiene como llaves los scid y los valores son una lista con todos los puntos de ese scid
24    mapa_scc=mp.newMap(200,
25                        maptype="PROBING",
26                        loadfactor=0.5,
27                        cmpfunction=compare_map)
28
29    lista_puntos=lista_eventos.ll.newList(datastructure="ARRAY LIST")
30    for punto in lt.iterator(keys):
31        entry=mp.get(mapa_idsc, punto)
32        scid= mp.getValue(entry)
33        if mp.contains(mapa_scc, scid):
34            entry= mp.get(mapa_scc, scid)
35            lista= mp.getValue(entry)
36            lt.addlast(lista, punto)
37            mp.put(mapa_scc, scid, lista)
38        else:
39            lista_puntos=lista_eventos.ll.newList(datastructure="ARRAY LIST")
40            lt.addlast(lista_puntos, punto)
41            mp.put(mapa_scc, scid, lista_puntos)
42
43    lista_final=ll.newList(datastructure="ARRAY LIST")
44    mapa=mp.newMap(200,
45                  maptype="PROBING",
46                  loadfactor=0.5,
47                  cmpfunction=compare_map)
48
49    keys= mp.keySet(mapa_scc)
50
51    for iden in lt.iterator(keys):
52        mapa=mp.newMap(20,
53                      maptype="PROBING",
54                      loadfactor=0.5,
55                      cmpfunction=compare_map)
56
57        lista= mp.put(mapa_scc, iden)
58        listar= mp.getValue(lista)
59        mp.put(mapa, "scid", iden)
60        mp.put(mapa, "accsize", lt.size(listar))
61        mp.put(mapa, "valores", lista)
62        lt.addlast(lista_final, mapa)
63
64    lista_final=sort(lista_final, 2)
65    lista_final= tree_print_ll(lista_final)
66    lista_final2=ll.newList(datastructure="ARRAY LIST")
67    for mapa in lt.iterator(lista_final):
68        lista= mp.get(mapa, "valores")
69        lista= mp.getValue(lista)
70        lista=sort(lista, 2)
71        minlon, maxlon= obtener_max_min_llon(lista)
72        mp.put(mapa, "maxlon", maxlon)
73        mp.put(mapa, "minlon", minlon)
74        num_wolf, wolf= obtener_num_wolf(control["wolves"], lista)
75        mp.put(mapa, "wolfetails", wolf)
76        mp.put(mapa, "wolfcount", num_wolf)
77        arcos, vertices, distancia= crear_arcos_vertices_distancia(grafo, lista)
78        mp.put(mapa, "nodes", vertices)
79        mp.put(mapa, "edges", arcos)
80        mp.put(mapa, "distance", distancia)
81        minlat, maxlat= obtener_max_min_lat(lista)
82        mp.put(mapa, "maxlat", maxlat)
83        mp.put(mapa, "minlat", minlat)
84        puntos= crear_str_puntos(lista)
85        mp.put(mapa, "nodesid", puntos)
86        lt.addlast(lista_final2, mapa)
87
88    # lista final 2
89
90    lista_final=[]
91    for mapa in lt.iterator(lista_final2):
92        lista2=[]
93        lista_puntos=mp.get(mapa, "valores")
94        lista_puntos= mp.getValue(lista_puntos)
95        for punto in lt.iterator(lista_puntos):
96            lista2.append(punto)
97        lista_final.append(lista2)
98
99    return gr.numVertices(grafo), gr.numEdges(grafo), cc.connectedComponents(search), lista_final2
100
```

## Descripción

Esta función reporta el efecto de los cambios en las condiciones climáticas en la movilidad de las manadas y en el territorio. Para esto primero crea un mapa con los eventos, dividiéndolos por fecha, luego obtiene los datos de ese rango de fechas. Luego filtra estos eventos por temperatura y crea el grafo apartir de estos datos. Luego usa el algoritmo de Kosaraju para obtener los componentes fuertemente conectados y crea un mapa con los datos, donde el SCCID es la llave y los valores son una lista con los puntos, al fina aplica DJK para obtener la distancia, los nodos y arcos de las manadas. Y cresa una lista de mapas donde cada mapa es una manada.

Entrada	control, fecha incial, fecha final, tem_min, tem_max
Salidas	Nodos, arcos, SCCs, lista de mapas
Implementado (Sí/No)	Si

## Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	Complejidad
Crear arbol eventos	O(n)



Obtener rango	$O(\log n)$
Filtrar temperatura	$O(n)$
Crear grafo	$O((v+e)\log v)$
Aplicar DJK	$O((v+e)\log v)$
Crear lista de mapas	$O(n^2(\log n))$
<b>TOTAL</b>	$O(n^2(\log n))$

## Pruebas Realizadas

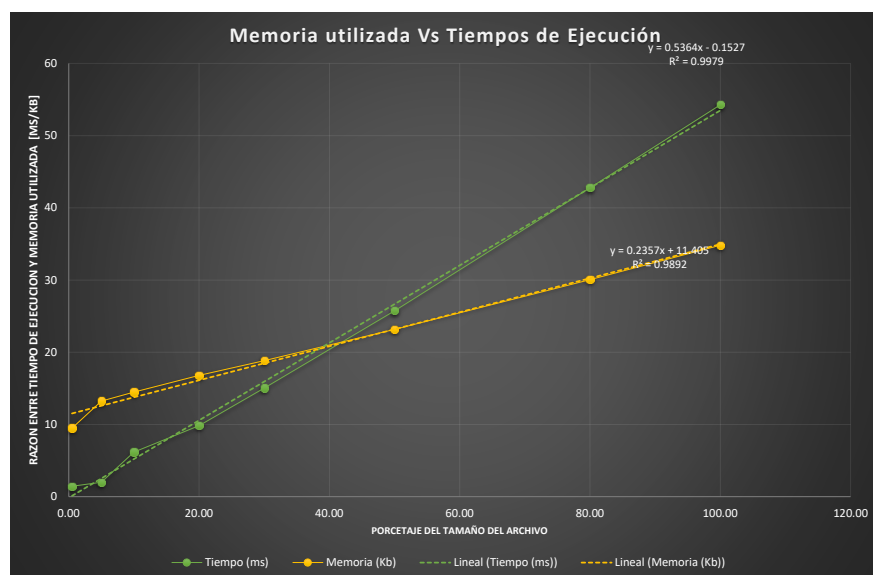
Las pruebas realizadas fueron realizadas en una maquina con las siguientes especificaciones. Los datos de entrada fueron el data structs del modelo.

Procesadores	<b>11th Gen Intel(R) Core(TM) i3-1115G4 @ 3.00GHz</b>
Memoria RAM	12 GB
Procesador	Core i3
Sistema Operativo	Windows 10

Entrada	Tiempo (s)
small	37.449
5 pct	29.157
10 pct	76.7
20 pct	127.722
30 pct	630.433
50 pct	1565.6521

## Graficas

Las gráficas con la representación de las pruebas realizadas.



## Análisis

A pesar de que la complejidad calculada fue de  $O(n)$ , en la gráfica se puede evidenciar que la función tiende a ser  $n \log n$ , lo que da por conclusión que la función en casos promedios suele ser  $n \log n$ .

## Requerimiento 8

```
if req8_bool:
    iterator = lt.iterator(lista_mapa)
    ln_i = lt.firstElement(lista_mapa)[0]
    lt_i = lt.firstElement(lista_mapa)[1]
    m = folium.Map(location=[lt_i, ln_i], zoom_start=12)
    trail = []
    for i in iterator:
        trail.append([i[1], i[0]])
    folium.PolyLine(trail).add_to(m)
    output_file = "req4.html"
    m.save(output_file)
```

## Descripción

La función está implementada en cada requerimiento, visto que hay un caso particular para cada uno, aunque es posible generalizar la serie de pasos que cumple. El llamado a la función solo cambia si genera mapas o no se debe llamar cada requerimiento para que se ejecute la función. En general, esta crea a partir de una lista con los puntos del recorrido una lista de coordenadas. Centra el mapa en la primera coordenada y crea una PolyLine con la lista de coordenadas. En algunos casos debe reconstruir las coordenadas a partir del nombre del punto y en otro asignarle un color a cada punto (req. 3 y 7).

<b>Entrada</b>	Lista de puntos
<b>Salidas</b>	Mapa html
<b>Implementado (Sí/No)</b>	Si.

## Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

<b>Pasos</b>	<b>Complejidad</b>
Acceder al primer elemento de la lista	$O(1)$
Crear mapa	$O(1)$
Recorrer lista	$O(n)$
Crear PolyLine	$O(n)$
<b>TOTAL</b>	<b><math>O(n)</math></b>

## Pruebas Realizadas

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

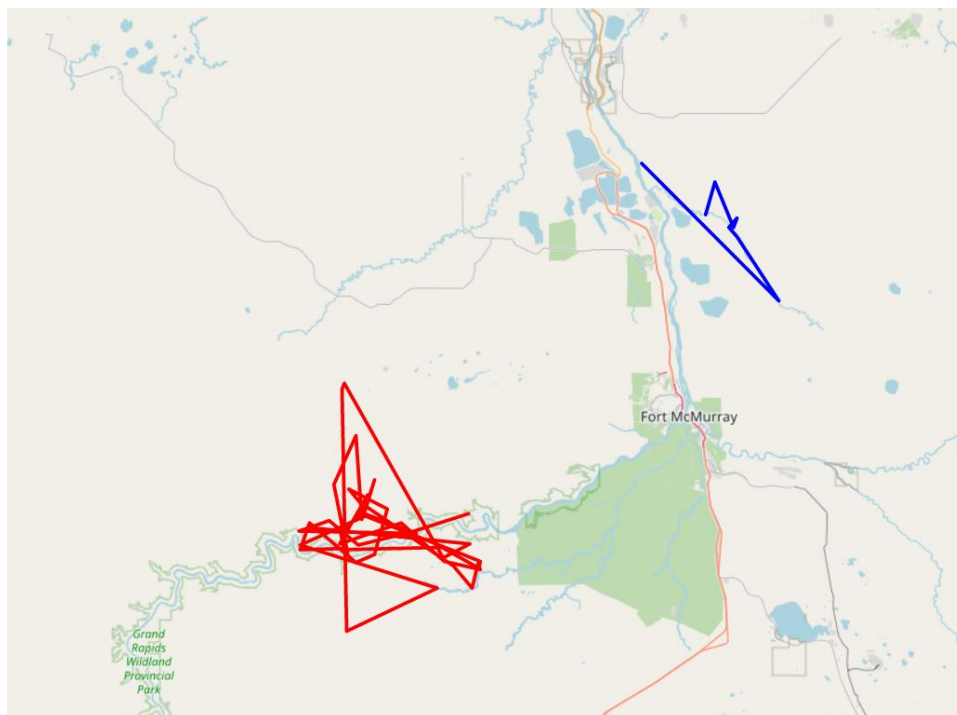
<b>Procesadores</b>	11th Gen Intel(R) Core(TM) i3-1115G4 @ 3.00GHz
<b>Memoria RAM</b>	12 GB
<b>Procesador</b>	Core i3
<b>Sistema Operativo</b>	Windows 10

Entrada	Tiempo (s)
small	37.449
5 pct	29.157
10 pct	76.7
20 pct	127.722
30 pct	430.433
50 pct	965.6521

### Salida (con el 100% de datos)

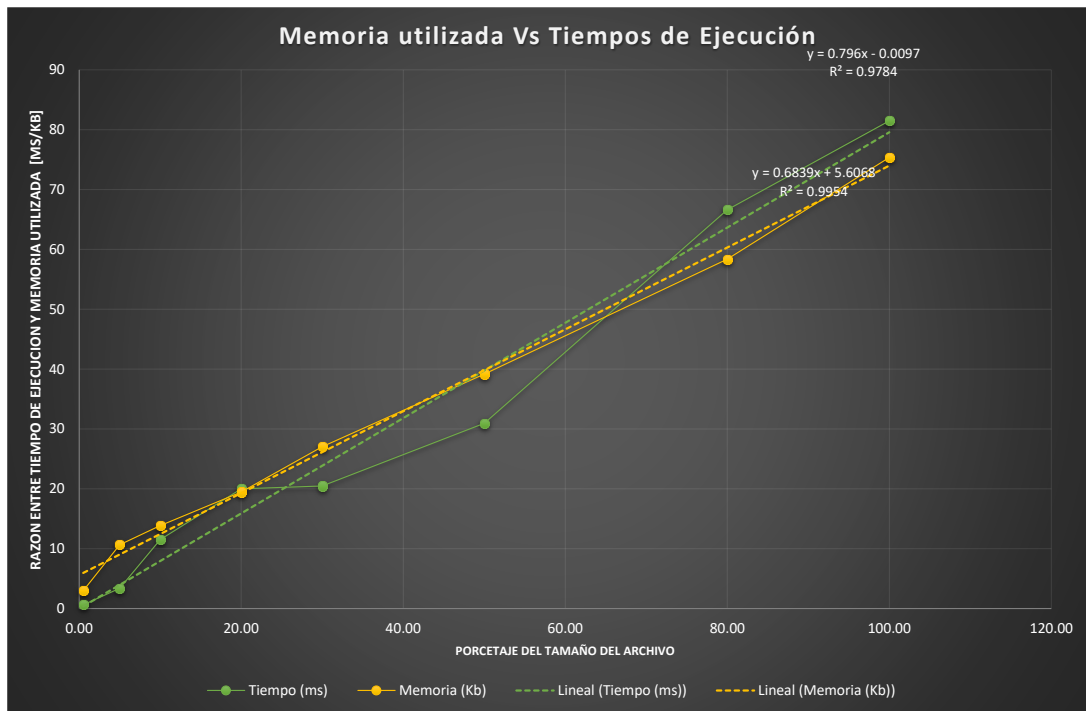
Las tablas con la recopilación de datos de las pruebas.

Req 6 small



### Graficas

Las gráficas con la representación de las pruebas realizadas.



## Análisis

El algoritmo se acerca a un tiempo de ejecución  $n \log n$  y se beneficia del uso de rangos pequeños.