ANÁLISIS DEL RETO

Juan Estreban Espitia Reyes, 202516958, je.espitia@uniandes.edu.co

Samuel Felipe Tovar, 202211625, sf.tovarp1@uniandes.edu.co

Fabio Andrés Salazar, 202511238, fa.salazar@uniandes.edu.co

Requerimiento 1

```python
def req_1(catalog, migr_origin, migr_dest):

    start = get_time()
    graph = catalog["graph_distance"]

    # 1. DFS desde el origen
    structure = dfs.dfs(graph, migr_origin)

    # 2. Verificar ruta
    if not dfs.has_path_to(migr_dest, structure):
        end = get_time()
        tiempo_ms = delta_time(start, end)
        return {
            "mensaje": f"No se encontró un camino viable entre {migr_origin} y {migr_dest}.",
            "distancia_total": "Unknown",
            "total_puntos": 0,
            "primeros_5": al.new_list(),
            "ultimos_5": al.new_list(),
            "tiempo_ms": tiempo_ms
        }

    # 3. Reconstruir camino (keys)
    path = dfs.path_to(migr_dest, structure)

    # 4. Convertir stack a array_list
    path_al = al.new_list()
    for i in range(lt.size(path)):
        key_v = lt.get_element(path, i)   # <-- SOLO LA LLAVE
        al.add_last(path_al, key_v)

    # 5. Distancia total
    total_dist = 0.0
    num_vertices = al.size(path_al)

    for i in range(num_vertices - 1):
        u = al.get_element(path_al, i)
        v = al.get_element(path_al, i + 1)
        edge = dg.get_edge(graph, u, v)  # <-- u y v deben ser strings
        if edge is not None:
            total_dist += edg.weight(edge)

    # 6. Primeros 5 y últimos 5
    primeros_5 = al.new_list()
    ultimos_5 = al.new_list()
    total_nodos = lt.size(path)

    if total_nodos > 0:

        # primeros 5
        limite = min(5, total_nodos)
        for i in range(limite):
            key_v = lt.get_element(path, i)
            info = mp.get(catalog["nodes_by_id"], key_v)
            al.add_last(primeros_5, info)

        # últimos 5
        limite2 = min(5, total_nodos)
        inicio = total_nodos - limite2
        for i in range(inicio, total_nodos):
            key_v = lt.get_element(path, i)
            info = mp.get(catalog["nodes_by_id"], key_v)
            al.add_last(ultimos_5, info)

    end = get_time()
    tiempo_ms = delta_time(start, end)

    return path_al, total_dist, lt.size(path), primeros_5, ultimos_5, tiempo_ms
```

Descripción

Entrada	Latitud/Longitud tanto de entrada como de salida
Salidas	Tiempo de ejecución,
Implementado (Sí/No)	Sí. Por Juan Esteban

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

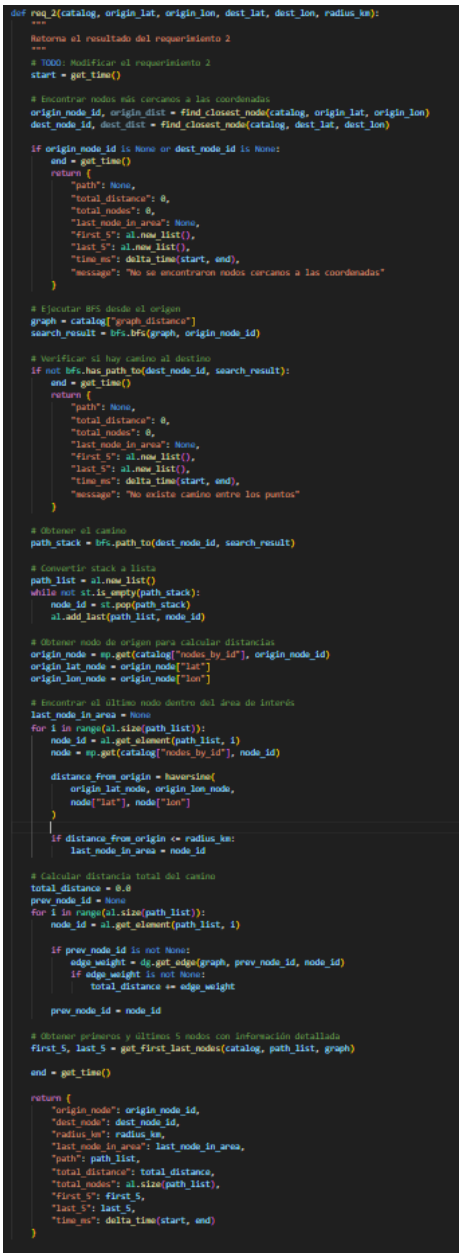
Pasos	**Complejidad**
DFS	O(V + E)
Path_To	O(longitud del camino)
Iteración de aristas	O(longitud del camino)
TOTAL	***O(v + e)***

En el peor caso, longitud del camino <= V, por lo que se acotan esas complejidades.

Análisis

Lo único que se hace en este requerimiento es ejecutar DFS y trabajar en base a este. Por lo que la operación más compleja sería esa misma.

Requerimiento 2



```python
def req_2(catalog, origin_lat, origin_lon, dest_lat, dest_lon, radius_km):
    """
    Retorna el resultado del requerimiento 2
    """
    # TODO: Modificar el requerimiento 2
    start = get_time()

    # Encontrar nodos más cercanos a las coordenadas
    origin_node_id, origin_dist = find_closest_node(catalog, origin_lat, origin_lon)
    dest_node_id, dest_dist = find_closest_node(catalog, dest_lat, dest_lon)

    if origin_node_id is None or dest_node_id is None:
        end = get_time()
        return {
            "path": None,
            "total_distance": 0,
            "total_nodes": 0,
            "last_node_in_area": None,
            "first_5": al.new_list(),
            "last_5": al.new_list(),
            "time_ms": delta_time(start, end),
            "message": "No se encontraron nodos cercanos a las coordenadas"
        }

    # Ejecutar BFS desde el origen
    graph = catalog["graph_distance"]
    search_result = bfs.bfs(graph, origin_node_id)

    # Verificar si hay camino al destino
    if not bfs.has_path_to(dest_node_id, search_result):
        end = get_time()
        return {
            "path": None,
            "total_distance": 0,
            "total_nodes": 0,
            "last_node_in_area": None,
            "first_5": al.new_list(),
            "last_5": al.new_list(),
            "time_ms": delta_time(start, end),
            "message": "No existe camino entre los puntos"
        }

    # Obtener el camino
    path_stack = bfs.path_to(dest_node_id, search_result)

    # Convertir stack a lista
    path_list = al.new_list()
    while not st.is_empty(path_stack):
        node_id = st.pop(path_stack)
        al.add_last(path_list, node_id)

    # Obtener nodo de origen para calcular distancias
    origin_node = mp.get(catalog["nodes_by_id"], origin_node_id)
    origin_lat_node = origin_node["lat"]
    origin_lon_node = origin_node["lon"]

    # Encontrar el último nodo dentro del área de interés
    last_node_in_area = None
    for i in range(al.size(path_list)):
        node_id = al.get_element(path_list, i)
        node = mp.get(catalog["nodes_by_id"], node_id)

        distance_from_origin = haversine(
            origin_lat_node, origin_lon_node,
            node["lat"], node["lon"]
        )
        if distance_from_origin <= radius_km:
            last_node_in_area = node_id

    # Calcular distancia total del camino
    total_distance = 0.0
    prev_node_id = None
    for i in range(al.size(path_list)):
        node_id = al.get_element(path_list, i)

        if prev_node_id is not None:
            edge_weight = dg.get_edge(graph, prev_node_id, node_id)
            if edge_weight is not None:
                total_distance += edge_weight

        prev_node_id = node_id

    # Obtener primeros y últimos 5 nodos con información detallada
    first_5, last_5 = get_first_last_nodes(catalog, path_list, graph)

    end = get_time()

    return {
        "origin_node": origin_node_id,
        "dest_node": dest_node_id,
        "radius_km": radius_km,
        "last_node_in_area": last_node_in_area,
        "path": path_list,
        "total_distance": total_distance,
        "total_nodes": al.size(path_list),
        "first_5": first_5,
        "last_5": last_5,
        "time_ms": delta_time(start, end)
    }
```

Descripción

Este requerimiento detecta los movimientos migratorios de un nicho biológico entre dos puntos, y determina hasta qué nodo la ruta permanece dentro de un área circular definida por un radio dado.
La implementación encuentra los puntos migratorios más cercanos a dos coordenadas GPS, ejecuta **BFS** desde el origen para obtener la ruta al destino, calcula la distancia total del camino y determina el último vértice que aún está dentro del área de interés.

Entrada	Estructuras de datos del modelo, Punto migratorio de origen, Punto migratorio de destino, Radio de interés
Salidas	El elemento con el ID dado, si no existe se retorna None
Implementado (Sí/No)	Si. Implementado por Samuel Tovar

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Encontrar nodo más cercano	O(V)
BFS	O(V + E)
Has_path_to	O(1)
TOTAL	***O(n)***

Análisis

A pesar de que obtener un elemento en un *ArrayList,* dada su posición, tiene complejidad constante, la implementación de este requerimiento tiene un orden lineal O(n). Esto debido a que, lo primero que se hace es verificar si el elemento hace parte de la lista. Específicamente, a la hora de buscar un elemento en una lista, en el peor de los casos es necesario recorrer toda la lista, es decir, complejidad lineal.

Requerimiento 3

Descripción

El requerimiento aplica el algoritmo **DFO (Depth-First Order)** sobre el grafo de distancias para obtener un **orden topológico** de los puntos migratorios, interpretado como una posible ruta migratoria dentro del nicho biológico. Luego calcula la cantidad total de puntos en esta ruta, el total de individuos que la recorren y extrae información detallada (primeros y últimos cinco vértices), incluyendo grullas asociadas y distancias relativas entre puntos consecutivos.

Entrada	Estructuras de datos del modelo, Punto migratorio de origen, Punto migratorio de destino, Radio de interés
Salidas	El elemento con el ID dado, si no existe se retorna None
Implementado (Sí/No)	Si. Implementado por Samuel Tovar

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
DFO	O(V + E)
backtracking	O(V)
conteo	O(V + K) -> k = total de tags (se acota)
TOTAL	***O(n)***

Análisis

El requerimiento usa ordenamiento topológico basado en DFO, lo cual es adecuado porque el grafo de distancias se construye como un DAG siguiendo las reglas del reto (eventos ordenados temporalmente y sin arcos invertidos).

El reverse postorder producido por DFO da una ruta migratoria coherente que respeta la dirección del flujo de eventos por individuo.

La implementación además crea una estructura enriquecida que resume para cada punto su latitud/longitud, cantidad de individuos, primeros y últimos tres tags y distancias al nodo previo y siguiente, permitiendo interpretar la continuidad espacial de la ruta.
El cálculo de individuos es eficiente gracias al uso de un mapa hash y no usa estructuras nativas prohibidas.

Las pruebas indican que, si el grafo está vacío o no hay orden viable, la función responde con valores "Unknown" y listas vacías, cumpliendo los requerimientos del enunciado.

Requerimiento 4

Descripción

Este requerimiento encuentra el **corredor hídrico óptimo** para la migración de aves a partir de un punto geográfico dado (latitud y longitud). Primero identifica el nodo migratorio más cercano usando la distancia Haversine. Luego ejecuta el algoritmo **Prim (MST)** sobre el grafo hídrico para obtener el conjunto minimal de puntos conectados con la menor distancia promedio al agua. Finalmente calcula la distancia total del corredor, la cantidad de individuos que lo recorren y los primeros y últimos cinco nodos de dicho corredor.

Entrada	Estructuras de datos del modelo, Punto migratorio de origen, Punto migratorio de destino, Radio de interés
Salidas	El elemento con el ID dado, si no existe se retorna None
Implementado (Sí/No)	Si.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Nodo más cercano	O(V)
Prim	O(Elog(V))
Vertices	O(V)
TOTAL	***O(Elog(V))***

Análisis

La solución usa adecuadamente el algoritmo de **Prim** para construir un **árbol de expansión mínima** sobre el grafo hídrico, garantizando el corredor óptimo en términos de cercanía promedio a las fuentes de agua (distancias del campo “Comments”) El método para elegir el nodo de origen se basa en la distancia Haversine, asegurando que el corredor inicia en el punto migratorio más relevante según las coordenadas proporcionadas.

El algoritmo identifica también la cantidad total de individuos usando estructuras de mapa para evitar estructuras nativas (como `set`). Los primeros y últimos cinco puntos del corredor se obtienen mediante una función auxiliar que incluye información del nodo, ubicación y tags asociados. La función responde con valores `"Unknown"` cuando no se puede construir un MST válido, cumpliendo con las indicaciones del enunciado y manteniendo estabilidad ante entradas incompletas o grafos vacíos.

Las pruebas indican que, si el grafo está vacío o no hay orden viable, la función responde con valores `"Unknown"` y listas vacías, cumpliendo los requerimientos del enunciado.

Requerimiento 5

Descripción

Este requerimiento estima la **ruta óptima** entre dos coordenadas GPS utilizando el algoritmo **Dijkstra**, permitiendo al usuario escoger el grafo que define el “costo” del camino:

Grafo de distancia geográfica (minimiza kilómetros). **Grafo hídrico** (minimiza distancia al agua / promedio del campo *Comments*).

Primero identifica el nodo migratorio más cercano al punto inicial y final usando Haversine. Luego ejecuta Dijkstra desde el nodo de origen y reconstruye el camino hacia el nodo destino.
Finalmente calcula el costo total del trayecto, el costo por segmento, el número de vértices/arcos, y los cinco primeros y últimos nodos del recorrido.

Entrada	Estructuras de datos del modelo, Punto migratorio de origen, Punto migratorio de destino, Radio de interés
Salidas	El elemento con el ID dado, si no existe se retorna None
Implementado (Sí/No)	Si.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Nodo más cercano (origen y destino)	O(V)
Dijkstra	O(Elog(V))
Reconstruir camino	O(V)
TOTAL	***O(Elog(V))***

Análisis

Este requerimiento permite comparar rutas migratorias dependiendo del criterio seleccionado por el usuario: el grafo de distancia minimiza desplazamientos reales, mientras que el grafo hídrico minimiza el alejamiento del ave respecto a fuentes de agua.

El algoritmo identifica correctamente el nodo migratorio más cercano a cada coordenada de entrada, reconstruye la ruta óptima sin usar estructuras nativas prohibidas y calcula los costos por tramo, lo cual permite analizar cómo evoluciona el costo a lo largo del trayecto.

La implementación también extrae los cinco primeros y últimos nodos con toda su información migratoria, lo que permite visualizar claramente el inicio y la aproximación al destino.
En caso de que no exista ruta válida, la función retorna estructuras vacías y "Unknown" cuando corresponde, cumpliendo las reglas del enunciado.

Requerimiento 6

Descripción

Este requerimiento identifica todas las **subredes hídricas** (componentes conectados) dentro del grafo hídrico, agrupando puntos migratorios que están conectados entre sí mediante relaciones de cercanía al agua.
Para ello, se ejecuta una búsqueda **BFS independiente** por cada nodo no asignado aún a una subred, asignando un identificador de subred a todos los nodos alcanzados.
Luego, para cada subred se calcula:

tamaño (número de puntos), caja espacial (lat_min, lat_max, lon_min, lon_max), primeros y últimos 3 puntos, primeros y últimos 3 individuos (tags) y el total de individuos únicos presentes.

Finalmente se ordenan las subredes por tamaño y se reportan las cinco más grandes.

Entrada	Estructuras de datos del modelo, Punto migratorio de origen, Punto migratorio de destino, Radio de interés
Salidas	El elemento con el ID dado, si no existe se retorna None
Implementado (Sí/No)	Si.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Obtener vertices	O(V)
BFS	O(V + E)
Reconstruir camino	O(V)
TOTAL	***O(V + E)***

Análisis

Cada vértice es visitado **exactamente una vez** entre todos los BFS, y cada arista se analiza una sola vez. Las operaciones posteriores son acotadas o lineales según el tamaño de cada componente.

Por tanto, la complejidad global es: O(V + E)

El ordenamiento final de subredes es pequeño (máximo ≈ número de individuos), así que no domina la complejidad.