ANÁLISIS DEL RETO 3

Juliana Rodríguez Morales – 202421552 – js.rodriguezm1234
Maria Clara Quijano - 202420069 - m.quijanoa
Juan Andrés Lozada - 202510410-j.lozadab

Requerimiento 1

Descripción

```
def req_1(catalog, lat_o, lon_o, lat_d, lon_d, grulla_id):
    grafo = catalog["grafo_migraciones"]
    vids = d.vertices(grafo)
    if vids is None or al.size(vids) == 0:
        return {"error":"Grafo vacío."}
    origen = None
    destino = None
    mo = float("inf")
    md = float("inf")
    i = 0
    while i < al.size(vids):
        vid = al.get_element(vids, i)
        info = d.get_vertex_information(grafo, vid)
        if info is not None and "location" in info and info["location"] is not None:
            loc = info["location"]
            do = h.haversine((lat_o, lon_o), loc)
            dd = h.haversine((lat_d, lon_d), loc)
            if do < mo:
                mo = do; origen = info
            if dd < md:
                md = dd; destino = info
        i = i + 1
    if origen is None or destino is None:
        return {"error":"No se encontraron origen/destino cercanos."}
    oid = origen["event_id"] if "event_id" in origen else "Unknown"
    did = destino["event_id"] if "event_id" in destino else "Unknown"
    search = dfs.dfs(grafo, oid)
    path = dfs.path_to(did, search)
    # construir nodos
```

```
nodes = []
total_dist = 0.0
idx = 0
npath = sl.size(path)
while idx < npath:
    nid = sl.get_element(path, idx)
    info = d.get_vertex_information(grafo, nid)
    lat = "Unknown"; lon = "Unknown"; conteo = "Unknown"
    p3 = ["Unknown","Unknown","Unknown"]
    u3 = ["Unknown","Unknown","Unknown"]
    if info is not None:
        if "location" in info and info["location"] is not None:
            lat = info["location"][0]; lon = info["location"][1]
        conteo = info["conteo"] if "conteo" in info else "Unknown"
        if "grullas" in info and al.size(info["grullas"])>0:
            ng = al.size(info["grullas"])
            ptemp = []
            utemp = []
            j = 0
            while j < min(3, ng):
                ptemp.append(al.get_element(info["grullas"], j))
                j = j+1
            k = ng - min(3, ng)
            while k < ng:
                utemp.append(al.get_element(info["grullas"], k))
                k = k+1
            while len(ptemp)<3:
                ptemp.append("Unknown")
            while len(utemp)<3:
```

```
            utemp.append("Unknown")
        p3 = ptemp
        u3 = utemp
    dist_next = None
    if idx < npath - 1:
        nxt = sl.get_element(path, idx+1)
        vert = d.get_vertex(grafo, nid)
        print (vert["adjacents"])
        print (nxt)
        if vert is not None and "adjacents" in vert and vert["adjacents"] is not None:
            w = m.get(vert["adjacents"], nxt)
            if w is not None:
                wf = float(w)
                dist_next = round(wf,2)
                total_dist = total_dist + wf
    nodes.append({"id":nid,"lat":lat,"lon":lon,"conteo":conteo,"primeros3":p3,"ultimos3":u3,"dist_next":dist_next})
    idx = idx + 1

total_n = len(nodes)
primeros5 = []
ultimos5 = []
i = 0
while i < min(5, total_n):
    primeros5.append(nodes[i]); i = i + 1
j = total_n - min(5, total_n)
while j < total_n:
    ultimos5.append(nodes[j]); j = j + 1
```

```
# primer nodo donde aparece la grulla
first = "Unknown"
i = 0
while i < total_n:
    if nodes[i]["primeros3"] is not None:
        k = 0
        while k < 3:
            if nodes[i]["primeros3"][k] == grulla_id:
                first = nodes[i]["id"]; k = 3; i = total_n
            k = k + 1
    i = i + 1

mensaje = ("El individuo aparece por primera vez en el nodo: " + str(first)) if first != "Unknown" else "Unknown"

return {"origen_id":oid,"destino_id":did,"mensaje_first_node":mensaje,
        "total_dist":round(total_dist,2),"total_nodos":total_n,"primeros5":primeros5,"ultimos5":ultimos5}
```

Entrada	• Punto migratorio de origen, definido por ser el nodo más cercano a la locación GPS especificada por el usuario (latitud-longitud) • Punto migratorio de destino, definido por ser el nodo más cercano a la locación GPS especificada por el usuario (latitud-longitud).
Salidas	Mensaje que indique el primer nodo (punto migratorio) del camino donde se encuentre el individuo especificado. La distancia de desplazamiento total que durará el camino entre el punto de origen y el de destino . El total de puntos que contiene el camino, Mostrar los cinco primeros y cinco últimos vértices (puntos migratorios) que definen la ruta (incluyendo el origen y de destino) con la siguiente información: El identificador del punto migratorio.

	La longitud y latitud del punto. El número de individuos (grullas) que por ese punto. Listado con los tres primeros y tres últimos identificadores de las grullas que transitan por el punto.La distancia de desplazamiento al siguiente vértice en la ruta (punto migratorio)
Implementado (Sí/No)	Si, por Juan Andres Lozada

Análisis de complejidad

1. Buscar nodo origen y destino más cercanos: O(V).

2. Ejecutar DFS desde el origen: O(V + E).

3. Comprobar si existe camino y reconstruirlo: O(P) donde P ≤ V.

4. Recorrer el camino y construir la info por vértice: O(P)

Resultado: O(V + E)

Análisis

La complejidad es O(V + E) porque el algoritmo ejecuta una búsqueda en profundidad (DFS) desde el punto de origen, visitando cada vértice y arista únicamente una vez. Además, la reconstrucción del camino y la extracción de la información asociada a cada nodo son operaciones lineales respecto al tamaño del recorrido.

Requerimiento 2

Descripción

Miraba dentro de un área entrada por parámetro los movimientos de las grullas entre dos puntos migratorios.

```python
def req_2(catalog, p_origen, p_destino, radio):
    """
    Retorna el resultado del requerimiento 2
    """
    # TODO: Modificar el requerimiento 2
    start=get_time()
    migraciones = catalog["grafo_migraciones"]
    vertices = catalog["vertices_llaves"]
    nodo_origen = None
    minO = 99999999999999
    nodo_destino = None
    minD = 99999999999999

    for i in vertices:
        nodo = d.get_vertex_information(migraciones, i)
        distancia = h.haversine((p_origen[0], p_origen[1]), nodo["location"])
        if distancia < minO:
            nodo_origen = i
            minO = distancia
        distancia = h.haversine((p_destino[0], p_destino[1]), nodo["location"])
        if distancia < minD:
            nodo_destino = i
            minD = distancia
    if nodo_origen is not None or nodo_destino is not None:
        caminosMigraciones = bfs.bfs(migraciones, nodo_origen)
    else:
        return "No se pudo determinar nodo origen o destino."
    camino = bfs.path_to(nodo_destino, caminosMigraciones)
    if camino is None:
```

```
if camino is None:
    return "No existe una ruta viable entre los puntos."

resultado = {}
dist_total = 0
ultimo = None
while not s.is_empty(camino):
    elem = s.pop(camino)
    vert = d.get_vertex_information(migraciones, elem)
    mapa_bfs = m.get(caminosMigraciones, elem)
    if mapa_bfs is None:
        continue
    dist_to = mapa_bfs["dist_to"]
    if dist_to <= radio:
        ultimo = vert
        resultado[vert["id"]] = {"Location": vert["location"],
            "Grullas": len(vert["grullas"]),
            "Primeros 3": vert["grullas"][0:3],
            "Últimos 3": vert["grullas"][-1:-4],
            "dist_to": dist_to}
    else:
        continue

r = {"Ultimo en radio: ":ultimo,
     "Distancia total: ": round(dist_total, 3),
     "Total nodos": len(resultado),
     "Camino":resultado}
end=get_time()
tiempo = delta_time(start, end)
return r, tiempo
```

Entrada	Punto migratorio de origen Punto migratorio de destino Radio del área de interés
Salidas	- Último nodo dentro del área de interés - Desplazamiento total del camino del origen al de llegada - Total de puntos del camino - Cinco primeros y últimos que muestren: ○ Identificador punto migratorio

	○ Longitud y latitud ○ 3 Primeros y Últimos id de grullas ○ Distancia al desplazamiento al siguiente vértice
Implementado (Sí/No)	Sí, por Juliana Rodriguez

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Primer recorrido for	O(V)
Creación del BFS	O(V+E)
While que desocupa la pila	O(n)
TOTAL	(V+(V+E)+n)

Análisis

Utiliza el algoritmo BFS para poder ir yendo por niveles y mirar aquellos dentro del área. Al ser BFS no importa mucho si es dirigido o no dirigido y si tiene o no pesos. Inicialmente se hizo un recorrido para poder encontrar el nodo de origen y destino de acuerdo con el vértice más cercano a las coordenadas básicas. Después se calcula el BFS para poder encontrar los vecinos por niveles, en este caso aves migratorias. Por último se hace un while para que desocupe la pila (sin embargo, como este ya se ha filtrado por los que están dentro del rango se toma otra variable. Como estos for's/while se hicieron por separado se suman ya que la otra no está ligada a la anterior dierectamente.

Requerimiento 3

Descripción

```python
def req_3(catalog):
    """
    Retorna el resultado del requerimiento 3
    """
    # TODO: Modificar el requerimiento 3
    nicho=catalog["grafo_migraciones"]
    visitados=m.new_map(d.order(nicho), 7)
    pila_dfs=s.new_stack()
    pila_top=s.new_stack()
    vertices=d.vertices(nicho)
    n=al.size(vertices)
    for i in range(n):
        ve=al.get_element(vertices,i)
        if m.get(visitados,ve) is None:
            s.push(pila_dfs,(ve,0))

    while not s.is_empty(pila_dfs):
        nodo,est=s.pop(pila_dfs)
        if est==0:
            if m.get(visitados,nodo) is None:
                m.put(visitados,nodo,{"marked":True,"edge_from":None})
                s.push(pila_dfs,(nodo,1))
                adyacentes=d.adjacent(nicho,nodo)
                if adyacentes is not None:
                    for j in range(al.size(adyacentes)-1,-1,-1):
                        vecino=al.get_element(adyacentes,j)
                        if m.get(visitados,vecino) is None:
                            s.push(pila_dfs,(vecino,0))
        else:
            s.push(pila_top,nodo)
    orden_top=[]
    while not s.is_empty(pila_top):
        orden_top.append(s.pop(pila_top))
    if len (orden_top)==0:
        return None

    dist=m.new_map(d.order(nicho),7)
    ant=m.new_map(d.order(nicho),7)
    for v in orden_top:
        m.put(dist,v,1)
        m.put(ant,v,None)
    mejor_fin=None
    mejor_long=0
```

```
    for u in orden_top:
        du=m.get(dist,u)
        adyacentes=d.adjacent(nicho,u)
        if adyacentes is not None:
            for j in range(al.size(adyacentes)):
                v=al.get_element(adyacentes,j)
                dv=m.get(dist,v)
                if dv is None:
                    dv=1
                    m.put(dist,v,dv)
                    m.put(ant,v,None)
                if du+2>dv:
                    m.put(dist,v,du+1)
                    m.put(ant,v,u)
        du_act=m.get(dist,u)
        if du_act>mejor_long:
            mejor_long=du_act
            mejor_fin=u

    if mejor_fin is None or mejor_long<2:
        return None
    camino_rev=[]
    actual=mejor_fin
    while actual is not None:
        camino_rev.append(actual)
        actual=m.get(ant,actual)

    camino=[]
    for i in range(len(camino_rev)-1,-1,-1):
        camino.append(camino_rev[i])

    tot_puntos=len(camino)

    individuos=set()
    for v_id in camino:
        info_v=d.get_vertex_information(nicho,v_id)
        lista_grullas=info_v["grullas"]
        for k in range(al.size(lista_grullas)):
            individuos.add(al.get_element(lista_grullas,k))
    tot_individuos=len(individuos)
    primeros=[]
    ultimos=[]
```

```
tot_puntos=len(camino)

individuos=set()
for v_id in camino:
    info_v=d.get_vertex_information(nicho,v_id)
    lista_grullas=info_v["grullas"]
    for k in range(al.size(lista_grullas)):
        individuos.add(al.get_element(lista_grullas,k))
tot_individuos=len(individuos)
primeros=[]
ultimos=[]

if tot_puntos<5:
    limite=tot_puntos
else:
    limite=5
for i in range(limite):
    v_id=camino[i]
    info_v=d.get_vertex_information(nicho,v_id)
    lat=info_v["location"][0]
    long=info_v["location"][1]
    datos={"Identificador único":info_v["event_id"],
           "Posición (lat, lon)":(round(lat,5),round(long,5)),
           "Fecha de creación":info_v["timestamp"],
           "Grullas (Tags)":info_v["grullas"]["elements"],
           "Conteo de eventos":info_v["conteo"]}
    primeros.append(datos)
for i in range(tot_puntos-limite,tot_puntos):
    v_id=camino[i]
    info_v=d.get_vertex_information(nicho,v_id)
    lat=info_v["location"][0]
    long=info_v["location"][1]
    datos={"Identificador único":info_v["event_id"],
           "Posición (lat, lon)":(round(lat,5),round(long,5)),
           "Fecha de creación":info_v["timestamp"],
           "Grullas (Tags)":info_v["grullas"]["elements"],
           "Conteo de eventos":info_v["conteo"]}
    ultimos.append(datos)
return tot_puntos,tot_individuos,primeros,ultimos
```

Entrada Catalogo, que contiene un grafo de migraciones	
Salidas - Indicación si existe ruta migratoria válida dentro del nicho biológico - Mostrar mensaje si no es DAG ○ Un ciclo - Mostrar mensaje si es DAG - Número total de puntos - Número total de individuos - Primeros cinco puntos migratorios de la ruta - Últimos cinco	
Implementado (Sí/No) sí, Maria Clara Quijano	

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**

DFS topológico	O(V+E)
Longest path en DAG	O(V+E)
Procesamiento del camino	O(L)
TOTAL	O(V+E)

Análisis

Utilizamos una pila para hacer el DFS y construir el orden topológico, usamos mapas para marcar visitados y almacenar distancias y otra información. Las listas nos sirvieron para recorreer vértices y adyacencias. Todo esto nos ayudó a que el algoritmo del camino más largo en un DAG se pudiera ejecutar en tiempo O(V+E)

Requerimiento 4

Descripción

El requerimiento 4 buscaba buscar un recorrido eficiente hacia las redes más cercanas fuentes hídricas a partir de un punto de origen dado.

```python
def req_4(catalog, p_origen):
    """
    Retorna el resultado del requerimiento 4
    """
    # TODO: Modificar el requerimiento 4
    start = get_time()
    hidrico = catalog["grafo_hidrico"]
    vertices = catalog["vertices_llaves"]
    nodo_origen = None
    min0 = 99999999999999

    for i in vertices:
        nodo = d.get_vertex_information(hidrico, i)
        distancia = h.haversine((p_origen[0], p_origen[1]), nodo["location"])
        if distancia < min0:
            nodo_origen = i
            min0 = distancia
    cam_efic = pr.prim(hidrico, nodo_origen)
    estructura_prim = pr.conexiones_costo(hidrico, cam_efic)
    #porque los prim deben tener siempre n-1 vértices, si no se cumple entonces po
    if len(cam_efic) != (d.order(hidrico) -1):
        return "No hay una red hídrica viable para el origen elegido"
    resultado = {"Total de puntos": len(cam_efic),
                 "Total de individuos": len(cam_efic),
                 "Distancia total": estructura_prim["peso_totoal"]}
    primeros = []
    ultimos = []
    nodos = estructura_prim["vertice"]
    for i in range(5):
        vertice = d.get_vertex_information(hidrico, nodos[i])
        p3 = []
        u3 = []
        keys = m.key_set(m.get(vertice["map_eventos"]))
        for j in range(3):
            p3.append(m.get(vertice["map_eventos"], al.get_element(keys, j)))
```

+

```
    for k in range(m.size(vertice["map_eventos"])-3, m.size(vertice["map_eventos"])):
        u3.append(m.get(vertice["map_eventos"], al.get_element(keys, k)))
    elemento = {"Id": vertice["id"],
                "Location": vertice["location"],
                "Grullas": len(vertice["grullas"]),
                "3 Primeros": p3,
                "3 Últimos": u3}
    primeros.append(elemento)

for i in range(len(nodos)-1, len(nodos)):
    vertice = d.get_vertex_information(hidrico, nodos[i])
    p3 = []
    u3 = []
    keys = m.key_set(m.get(vertice["map_eventos"]))
    for j in range(3):
        p3.append(m.get(vertice["map_eventos"], al.get_element(keys, j)))

    for k in range(m.size(vertice["map_eventos"])-3, m.size(vertice["map_eventos"])):
        u3.append(m.get(vertice["map_eventos"], al.get_element(keys, k)))
    elemento = {"Id": vertice["id"],
                "Location": vertice["location"],
                "Grullas": len(vertice["grullas"]),
                "3 Primeros": p3,
                "3 Últimos": u3}
    ultimos.append(elemento)
end = get_time()
tiempo = delta_time(start, end)
return resultado, tiempo
```

Entrada	Punto migratorio de origen

Salidas	- Total de puntos en una ruta migratoria
	- Total de individuos que usan la ruta migratoria
	- Distancia total
	- Mejores 5 y últimos 5 puntos migratorios:
	  ○ Identificador del punto migratorio original
	  ○ Longitud y latitud
	  ○ Número de grullas que transitan cerca
	  ○ 3 primeros y últimos id de grullas

Implementado (Sí/No)	Sí, por Juliana Rodríguez

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Primer recorrido for para hallar vértice	O(V)
Ejecución del Prim (función hecha por nosotros)	O(ELogV)

Procesamiento de conexiones y conteo	O(V)
Construcción primeros y últimos	O(n)
TOTAL	O(n+V+ELogV)

Análisis

Para poder hallar el vértice más cercano a los puntos dados se hizo un recorrido por cada uno para encontrar el de menor distancia, generando O(V) de complejidad. A partir de allí se hizo la función prim con el propósito de llegar a todos los vértices de forma eficiente. Ligado a esto se hizo un procesamiento de conexiones para llegar a los recorridos de cada vértice desde el original y el total de distancia recorrida y finalmente se organizó la información para su presentación.

Requerimiento 5

Descripción

```python
def req_5(catalog, lat_o, lon_o, lat_d, lon_d, grafo_tipo):
    # seleccionar grafo
    if grafo_tipo == "hidrico":
            grafo = catalog["grafo_hidrico"]
    else:
            grafo = catalog["grafo_migraciones"]
    vids = d.vertices(grafo)
    if vids is None or al.size(vids) == 0:
        return {"error":"El grafo no tiene vértices. Ejecute load_data."}

    # encontrar origen y destino más cercanos
    i = 0
    nvid = al.size(vids)
    nodo_or = None
    nodo_de = None
    mo = float("inf")
    md = float("inf")
    while i < nvid:
        vid = al.get_element(vids, i)
        info = d.get_vertex_information(grafo, vid)
        if info is not None and "location" in info and info["location"] is not None:
            loc = info["location"]
            do = h.haversine((lat_o, lon_o), loc)
            dd = h.haversine((lat_d, lon_d), loc)
            if do < mo:
                mo = do; nodo_or = info
            if dd < md:
                md = dd; nodo_de = info
        i = i + 1
    if nodo_or is None or nodo_de is None:
        return {"error":"No se encontraron nodos origen/destino cercanos."}
    source = nodo_or["event_id"] if nodo_or is not None else "Unknown"
    target = nodo_de["event_id"] if nodo_de is not None else "Unknown"
    # verificar que vertices existan
```

```
if not d.contains_vertex(grafo, source):
    return {"error":"Nodo origen no existe en grafo: " + str(source)}
if not d.contains_vertex(grafo, target):
    return {"error":"Nodo destino no existe en grafo: " + str(target)}
# Dijkstra
aux = djk.dijkstra(grafo, source)
if aux is None:
    return {"error":"Dijkstra falló con el nodo origen."}
if not djk.has_path_to(target, aux):
    return {"error":"No existe un camino viable entre los puntos."}
stack_path = djk.path_to(target, aux)
if stack_path is None:
    return {"error":"No se pudo reconstruir el camino."}
# convertir stack/estructura a lista normal path_list (0-based)
path_list = []
if "elements" in stack_path:
    k = 0
    while k < al.size(stack_path):
        path_list.append(al.get_element(stack_path, k))
        k = k + 1
elif "first" in stack_path:
    node = stack_path["first"]
    while node is not None:
        path_list.append(node["info"])
        node = node["next"]
else:
    return {"error":"Estructura de path desconocida."}
# asegurar orden source->...->target
if len(path_list) > 0:
    if path_list[0] != source and path_list[-1] == source:
        # invertir
        path_list = list(reversed(path_list))
```

```
camino = []
total_cost = djk.dist_to(target, aux) if hasattr(djk, "dist_to") else 0.0
# como fallback sumar pesos
total_cost = float(total_cost) if total_cost is not None else 0.0
idx = 0
npath = len(path_list)
while idx < npath:
    nid = path_list[idx]
    info = d.get_vertex_information(grafo, nid)
    lat = "Unknown"; lon = "Unknown"; conteo = "Unknown"
    p3 = ["Unknown","Unknown","Unknown"]; u3 = ["Unknown","Unknown","Unknown"]
    if info is not None:
        if "location" in info and info["location"] is not None:
            lat = info["location"][0]; lon = info["location"][1]
        conteo = info["conteo"] if "conteo" in info else "Unknown"
        if "grullas" in info and al.size(info["grullas"]) > 0:
            ng = al.size(info["grullas"])
            # primeros3
            ptemp = []
            j = 0
            lim = min(3, ng)
            while j < lim:
                ptemp.append(al.get_element(info["grullas"], j))
                j = j + 1
            # ultimos3
            utemp = []; start = ng - min(3, ng); k = start
            while k < ng:
                utemp.append(al.get_element(info["grullas"], k))
                k = k + 1
            while len(ptemp) < 3: ptemp.append("Unknown")
            while len(utemp) < 3: utemp.append("Unknown")
            p3 = ptemp; u3 = utemp
    dist_next = None
    if idx < npath - 1:
        nxt = path_list[idx + 1]
```

```
    dist_next = None
    if idx < npath - 1:
        nxt = path_list[idx + 1]
        edges_map = d.edges_vertex(grafo, nid)
        if edges_map is not None:
            edge_obj = m.get(edges_map, nxt)
            if edge_obj is not None and "weight" in edge_obj:
                wf = float(edge_obj["weight"])
                dist_next = round(wf, 2)
    nodo_info = {
        "id": nid,
        "lat": lat,
        "lon": lon,
        "conteo": conteo,
        "primeros3": p3,
        "ultimos3": u3,
        "dist_next": dist_next
    }
    camino.append(nodo_info)
    idx = idx + 1
total_nodos = len(camino)
total_segmentos = total_nodos - 1 if total_nodos > 0 else 0
total_cost = round(total_cost, 2)
return {"origen_id": source, "destino_id": target, "total_cost": total_cost,
        "total_nodos": total_nodos, "total_segmentos": total_segmentos, "camino": camino}
```

Entrada	•Punto migratorio de origen, dado por su cercania al punto GPS especificado por el usuario (latitud: longitud). •Punto migratorio de destino, dado por su cercania al punto GPS especificado por el usuario (latitud: longitud). •Selección entre el grafo por distancia de desplazamiento o el grafo por distancias a fuentes hídricas.
Salidas	El costo total que tomará el individuo (sea distancia de desplazamiento o distancia a fuentes hídricas) entre el punto de origen y el de destino, •El total de puntos que contiene el camino (vértices) •El total de segmentos que conforman la ruta identificada (arcos) •Mostrar los cinco primeros y cinco últimos vértices (puntos migratorios) que definen la ruta (incluyendo el origen y de destino) con la siguiente información ◯ El identificador del migratorio ◯ La longitud y latitud del punto. ◯ E número de individuos (grullas) que transitan por ese punto, ◯ Listado con los tres primeros y tres últimos identificadores de las grullas que transitan por e punto. ◯ La distancia al siguiente vértice en la ruta (punto migratorio)
Implementado (Sí/No)	Si, por Juan Andres Lozada

Análisis de complejidad

1. Buscar nodo origen y destino más cercanos: O(V)

2. Ejecutar Dijkstra desde el origen:O(E log V)

3. Comprobar existencia de camino y reconstruirlo: O(P)

4. Recorrer la ruta y construir la info por vértice: O(P)

5. Calcular costo total: O(P)

Resultado: O((V + E) log V))

Análisis

La complejidad es O(E log V) debido al uso del algoritmo de Dijkstra con una cola de prioridad. Cada arista puede relajar su peso una vez, lo que implica operaciones logarítmicas en el heap. La construcción del camino y la recolección de datos sobre los vértices también toman tiempo lineal en el tamaño del recorrido, sin alterar el costo dominante.

Requerimiento 6

```python
def req_6(catalog):
    """
    Retorna el resultado del requerimiento 6
    """
    # TODO: Modificar el requerimiento 6
    grafo=catalog["grafo_hidrico"]
    comp_vert=m.new_map(d.order(grafo),7)
    componentes={}
    comp_id=0
    vertices=d.vertices(grafo)
    n=al.size(vertices)
    for i in range(n):
        v=al.get_element(vertices,i)
        if m.get(comp_vert,v) is None:
            comp_id+=1
            q_nodes=q.new_queue()
            q.enqueue(q_nodes,v)
            m.put(comp_vert,v,comp_id)
            lista_vertices=[]
            grullas_subred=set()

            info_v=d.get_vertex_information(grafo,v)
            lat,lon=info_v["location"]
            min_lat=lat
            max_lat=lat
            min_lon=lon
            max_lon=lon
            while not q.is_empty(q_nodes):
                x=q.dequeue(q_nodes)
                lista_vertices.append(q.dequeue(x))
                info_x=d.get_vertex_information(grafo,x)
                latx,lonx=info_x["location"]
                if latx<min_lat:
                    min_lat=latx
                if latx>max_lat:
                    max_lat=latx
                if lonx<min_lon:
                    min_lon=lonx
                if lonx>max_lon:
                    max_lon=lonx
                tags=info_x["grullas"]
                for t in range (al.size(tags)):
                    grullas_subred.add(al.get_element(tags,t))
```

```
                max_lon=lonx
            tags=info_x["grullas"]
            for t in range (al.size(tags)):
                grullas_subred.add(al.get_element(tags,t))
            ady=d.adjacent(grafo,x)
            if ady is not None:
                for j in range(al.size(ady)):
                    w=al.get_element(ady,j)
                    if m.get(componentes,w) is None:
                        m.put(comp_vert,w,comp_id)
                        q.enqueue(q_nodes,w)
        componentes[comp_id]={"id": comp_id,"vertices":lista_vertices,"min_lat":min_lat,"max_lat":max_lat,"min_lon":min_lo
total_subredes=len(componentes)
if total_subredes==0:
    return 0,[]
lista_comp=[]
for cid in componentes:
    comp=componentes[cid]
    tam=len(comp["vertices"])
    comp['tam']=tam
    lista_comp.append(comp)
lista_comp.sort()
if len(lista_comp)>5:
    lista_comp=lista_comp[:5]
resultado=[]
for c in lista_comp:
    verts=comp["vertices"]
    tam=comp["tam"]
    lista_ordenada=[]
    for v in verts:
        info_v=d.get_vertex_information(grafo,v)
        lista_ordenada.append((info_v['timee_dt'],v))
    lista_ordenada.sort()

    puntos_prim=[]
    if tam>=3:
        limite_p=3
    else:
        limite_p=tam
    for i in range(limite_p):
        nd,vid=lista_ordenada[i]
        info_v=d.get_vertex_information(grafo,vid)
```

```
    for i in range(limite_p):
        nd,vid=lista_ordenada[i]
        info_v=d.get_vertex_information(grafo,vid)
        lat,lon=info_v["location"]
        puntos_prim.append({"id":info_v["event_id"],"lat":round(lat,5),"lon":round(lon,5)})

    puntos_ult=[]
    for i in range(tam-limite_p,tam):
        nd,vid=lista_ordenada[i]
        info_v=d.get_vertex_information(grafo,vid)
        lat,lon=info_v["location"]
        puntos_ult.append({"id":info_v["event_id"],"lat":round(lat,5),"lon":round(lon,5)})

    lista_grullas=sorted(list(comp["grullas"]))
    total_grullas=len(lista_grullas)
    if total_grullas>=3:
        limite_g=3
    else:
        limite_g=total_grullas
    grullas_prim=lista_grullas[:limite_g]
    if limite_g>0:
        grullas_ult=lista_grullas[-limite_g:]
    resumen={"id_subred":comp["id"],
             "total_puntos":tam,
             "min_latitud":round(comp["min_lat"],5),
             "max_latitud":round(comp["max_lat"],5),
             "min_longitud":round(comp["min_lon"],5),
             "max_longitud":round(comp["max_lon"],5),
             "total_grullas":total_grullas,
             "grullas_prim":grullas_prim,
             "grullas_ult":grullas_ult,
             "puntos_prim":puntos_prim,
             "puntos_ult":puntos_ult}
    resultado.append(resumen)
return total_subredes,resultado
```

Descripción

Identificar las subredes hídricas dentro del nicho biológico según los puntos migratorios relacionados con fuentes hídricas.

Entrada	Catalogo con el graafo hídrico
Salidas	- Cantidad total de subredes hídricas - cinco subredes hídricas más graandes y su información -
Implementado (Sí/No)	Sí, por Maria Clara Quijano

Análisis de complejidad

Pasos	**Complejidad**
Recorrer todos los vértices	O(V)
BFS	O(V+E)
Ordenar subredes	O(VlogV)

TOTAL	***O(V+E+VlogV)***

Análisis

Utilizamos una cola para realizar BFS en cada nodo no visitados. Los mapas fueron usados para registrar los nodos que ya habían sido asignados y la información de cada subred. Las listas nos sirvieron para almacenar los nodos que pertenecen a cada componente y ordenarlas por tamaño. Logramos tener una complejidad de O(V+E) debido a la adecuada elección de estructuras de datos.