Estructuras de Datos y Algoritmos: Reto 04

Juan Sebastián Ortega (Estudiante 1- js.ortergar1@uniandes.edu.co - 202021703) y Yesid Camilo Almanza (Estudiante 2 - y.almanza@uniandes.edu.co - 201921773)

Departamento de Ingeniería de Sistemas y Computación

UNIVERSIDAD DE LOS ANDES

7 de junio de 2021

I. Introducción

CON el objetivo de recrear la red global de cables submarinos que lleva al internet al rededor del mundo, se planteó adaptar esta red a una estructura tipo de **grafo** que pueda ser moldeada y manipulable en *Python*. Para la construcción de este modelo se hizo necesaria la carga inicial de tres archivos: connections.csv, landing_points.csv y countries.csv. El primero de estos esta encargado de mostrar los caminos o aristas entre landing points, el segundo contiene información adicional de los landing points y el tercero incluye información de las capitales de cada país representado en la base de datos. Es importante aclarar que de un mismo landing point parten múltiples cables, por lo que se optó en tomar como vértice la estructura *landing_point-cable*, lo cual será explicado en detalle más adelante.

Requerimientos a cumplir:

1. Encontrar la cantidad de clústeres (componentes conectados) dentro de la red de cables submarinos y si dos landing points pertenecen o no al mismo clúster.

2. Encontrar el (los) landing point(s) que sirven como punto de interconexión a más cables en la red.

3. Encontrar la ruta mínima en distancia para enviar información entre dos países, los puntos de origen y destino serán los landing point de la ciudad capital.

4. Identificar la infraestructura crítica para poder garantizar el mantenimiento preventivo del mismo. Para tal fin se requiere que se identifique la red de expansión mínima en cuanto a distancia que pueda darle cobertura a la mayor cantidad de landing point de la red.

5. Conocer el impacto que tendría el fallo de un determinado landing point que afecta todos los cables conectados al mismo. Para tal fin se requiere conocer la lista de países que podrían verse afectados al producirse una caída en el proceso de comunicación con dicho landing point.

6. Conocer el ancho de banda máximo que se puede garantizar para la transmisión a un servidor ubicado en el país A desde cada uno de los países conectados a un determinado cable. (BONO)

7. Encontrar la ruta mínima en número de saltos para enviar información entre dos direcciones IP dadas. (BONO)

8. Graficar en un mapa los resultados de cada uno de los requerimientos anteriormente enunciados. (BONO)

En el momento de cumplir con los requerimientos se tuvo en mente la complejidad temporal que iba a adquirir el algoritmo en cada uno de estos, de forma que se tomaron las siguientes decisiones para optimizar la eficiencia:

- Todos los ordenamientos se realizarán mediante la aplicación de **MERGE SORTING** puesto que es el algoritmo de ordenamiento más efectivo dentro de la *DISClib* de acuerdo a pruebas de eficiencia realizadas previamente.

- Todas las listas serán instanciadas como **ARRAY_LIST** debido a que se prioriza la utilización de las funciones $lt.addLast()$ y $lt.getElement()$ las cuales cuentan con una complejidad de $O(K)$ en este tipo de estructura.

- Todos los mapas de Hash serán creados de tipo **PROBING** y se mantendrá el factor de carga óptimo por defecto que ya incluye la *DISClib*. Esto se hace con el objetivo de mantener una estructura simple que a su vez tenga la menor probabilidad de encontrar colisiones por lo que se busca reducir el tiempo gastado en la función *rehash*.

II. Funciones Principales

El algoritmo de la aplicación fue diseñado de tal forma que los diferentes requerimientos comparten las mismas funciones (en su mayoría). Por lo tanto, antes de realizar el análisis general de la complejidad temporal de cada requerimiento es necesario analizar cada función principal individualmente.

Las funciones que se mostrarán a continuación se encuentran ubicadas en el *model.py* y son las encargadas de recorrer y obtener información respecto a los grafos mediante algoritmos especializados.

II-A. Funciones de aplicación de algoritmos

- getCluster()

```
def getCluster(analyzer):

    cluster_data = scc.KosarajuSCC(analyzer["connections"])

    return cluster_data
```

Como se puede observar, esta función únicamente utiliza el algoritmo de Kosaraju para encontrar los componentes fuertemente conectados dentro del grafo. De acuerdo a la teoría la complejidad espacial de este algoritmo es de **O(V)** y su complejidad temporal es de **O(V+E)**.

- minimumCountryRoute()

```
def minimumCountryRoute(analyzer, country_A, country_B):

    graph = analyzer["connections"]
    country_map = analyzer["countries"]
    capital_name_map = analyzer["info_landing_id"]

    capital_city_A_vertex = str(me.getValue(m.get(country_map, country_A))
    ["capital_id"])
    capital_city_B_vertex = str(me.getValue(m.get(country_map, country_B))
    ["capital_id"])

    dijkstra_search = djk.Dijkstra(graph, capital_city_A_vertex)

    if djk.hasPathTo(dijkstra_search, capital_city_B_vertex):

        dijkstra_cost = djk.distTo(dijkstra_search, capital_city_B_vertex)
        dijkstra_route = djk.pathTo(dijkstra_search, capital_city_B_vertex)

        return (dijkstra_cost, dijkstra_route)

    else:

        return None
```

En el caso de esta función se realizan múltiples consultas menores y accesos a mapas de hash con complejidad O(K) por lo que serán ignorados en el análisis general. Entonces la complejidad de esta función radica en su uso del algoritmo de Dijkstra que según la teoría tiene una complejidad espacial de **O(V + E)** y una complejidad temporal de **O((V+E) log V)** ya que el grafo está creado con una lista de adyacencia.

- getMST()

```
def getMST(analyzer):

    vertex_list = m.keySet(analyzer["landing_points"])
    vertex_amount = lt.size(vertex_list)


    MST = prim.PrimMST(analyzer["connections"])
    MST_tree = MST["mst"]

    MST_weight = prim.weightMST(analyzer["connections"], MST)
    MST_connected_nodes = MST_tree["size"]
    MST_largest_branch = MST_connected_nodes - 1

    #TODO
    return (MST_weight, MST_connected_nodes, MST_largest_branch)
```

En esta función, además de realizarse múltiples consultas despreciables, se utiliza el algoritmo de prim. Este algoritmo permite obtener un **Minimum Spanning Tree** a partir de un grafo dado. De acuerdo con la teoría la complejidad espacial de este algoritmo es de **O(V)** y su complejidad temporal es de **O(E log V)**.

II-B. Funciones de consulta auxiliares

- getClusterSize()

```
def getClusterSize(cluster):

    cluster_number = scc.connectedComponents(cluster)
    return cluster_number
```

Puesto que esta es una función de consulta simple sobre un componente ya creado, su complejidad temporal es constante y su uso de la memoria es mínimo, por lo que puede ser ignorada.

- stronglyConnected()

```
def getStronglyConnected(analyzer, cluster, landing_point_1, landing_point_2):

    strongly_connected = scc.stronglyConnected(cluster,
    landing_point_1, landing_point_2)

    return strongly_connected
```

Puesto que esta es una función de consulta simple sobre un componente ya creado, su complejidad temporal es constante y su uso de la memoria es mínimo, por lo que puede ser ignorada.

- getLandingPointConnections()

```
def getLandingPointConnections(analyzer):

    vertex_map = analyzer["landing_points"]
    vertex_list = m.keySet(vertex_map)
    landing_point_list = lt.newList(datastructure = "ARRAY_LIST")

    for vertex_key in lt.iterator(vertex_list):

        temp_list = lt.newList(datastructure = "ARRAY_LIST")
        vertex_value = lt.size(me.getValue(m.get(vertex_map, vertex_key)))

        lt.addLast(temp_list, vertex_key)
        lt.addLast(temp_list, vertex_value)

        lt.addLast(landing_point_list, temp_list)

    sorted_vertex_list = mergesort.sort(landing_point_list, cmpnumberofcables)

    top_10_vertex_list = lt.subList(sorted_vertex_list, 1, 10)
    top_10_vertex_list_final = lt.newList(datastructure = "ARRAY_LIST")

    for top_vertex in lt.iterator(top_10_vertex_list):

        top_vertex_id = int(lt.getElement(top_vertex, 1))

        top_vertex_info = me.getValue(m.get(analyzer["info_landing_id"], top_vertex_id))
        top_vertex_info["connected_cables"] = lt.getElement(top_vertex, 2)

        lt.addLast(top_10_vertex_list_final, top_vertex_info)

    #SE RETORNA UNA LISTA (TIPO LST) QUE CONTIENE 10 DICCIONARIOS
    return top_10_vertex_list_final
```

En este caso para encontrar el número de cables conectados a cada vertice, se hace necesario iterar a través de todos ellos. Por lo tanto, la complejidad de esta función es de **O(V)**. Sin embargo, también se aplica merge sort, por lo que se agrega la complejidad temporal **O(V log V)**. En cuanto a la complejidad espacial, se hace necesario hacer múltiples sub-listas de la información, pero la más grande tiene el tamaño de **O(V)**.

- getAdjacentVertices():

```
def getAdjacentVertices(analyzer, landing_point):

    graph = analyzer["connections"]

    landing_point_id = me.getValue(m.get(analyzer["info_landing_name"], landing_point))["land

    vertex_list = me.getValue(m.get(analyzer["landing_points"], landing_point_id))

    adjacent_list = lt.newList(datastructure = "ARRAY_LIST")

    for vertex in lt.iterator(vertex_list):

        vertex_name = landing_point_id + "-" + vertex

        temp_list = gr.adjacents(graph, vertex_name)
```

```
        lt.addLast(adjacent_list, temp_list)

    return adjacent_list
```

Esta función de consulta necesita iterar a través de todos los vértices para encontrar sus adyacentes y posteriormente agregarlos a una lista con el mismo tamaño. Por lo tanto, la complejidad espacial es de **O(V)** al igual que la temporal.

- getAdjacentCountries()

```
def getAdjacentCountries(analyzer, adjacent_vertices, landing_point):

    landing_map = analyzer["info_landing_id"]

    landing_map_name = analyzer["info_landing_name"]

    adjacent_countries = m.newMap(numelements=260,
                                  maptype='PROBING')

    adjacent_countries_list = lt.newList(datastructure = "ARRAY_LIST")


    for cable in lt.iterator(adjacent_vertices):

        for vertex in lt.iterator(cable):

            adjacent_landing = int(vertex.split("-")[0])

            adjacent_info = me.getValue(m.get(landing_map, adjacent_landing))

            if "name" in adjacent_info.keys():
                #Then the adjacent cable belongs to a normal city
                adjacent_country = adjacent_info["name"].split(",")[-1]

            else:
                #The cable is connected to a capital
                adjacent_country = adjacent_info["CountryName"]

            if not m.contains(adjacent_countries, adjacent_country):

                actual_landing_point_lat = float(me.getValue(m.get
                (landing_map_name,
                landing_point))["latitude"])
                actual_landing_point_lon = float(me.getValue(m.get
                (landing_map_name,
                landing_point))["longitude"])

                adjacent_country_lat = float(me.getValue(m.get(analyzer
                ["countries"],
                adjacent_country))["CapitalLatitude"])
                adjacent_countries_lon = float(me.getValue(m.get(analyzer
                ["countries"],
                adjacent_country))["CapitalLongitude"])

                adjacent_country_distance = haversine(actual_landing_point_lat,
                actual_landing_point_lon, adjacent_country_lat,
                adjacent_countries_lon)

                m.put(adjacent_countries, adjacent_country,
                adjacent_country_distance)


    for key in lt.iterator(m.keySet(adjacent_countries)):

        value = me.getValue(m.get(adjacent_countries, key))

        mini_list = lt.newList(datastructure = "ARRAY_LIST")
        lt.addLast(mini_list, key)
        lt.addLast(mini_list, value)

        lt.addLast(adjacent_countries_list, mini_list)


    return adjacent_countries_list
```

Esta función pareciera tener una complejidad cuadrática con relación a los vértices pero en verdad es mucho más simple de lo que parece. El primer *for loop* es utilizado para iterar a través de todos los landing points y luego a través de todos los cables conectados a cada landing point, como se estableció en el inicio cada vertice es una combinación entre landing point y cable específico, por lo tanto se estaría iterando únicamente sobre los vertices, generando una complejidad temporal de **O(N)**. En el peor de los casos (donde todos los vertices estén conectados de forma directa a uno solo), la complejidad espacial del mapa formado también sería O(N). El segundo for loop puede ser ignorado puesto que únicamente puede iterar sobre un máximo constante de 260 países.

II-B1. Funciones de ordenamiento:

- sortAdjacentCountries()

```
def sortAdjacentCountries(adjacent_countries):

    sorted_list = mergesort.sort(adjacent_countries, cmpCountriesDist)

    return sorted_list
```

Finalmente se tiene una función de ordenamiento que utiliza Merge Sort. Como es bien sabido la complejidad espacial de este algoritmo de ordenamiento es de **O(V)** mientras que su complejidad temporal es de **O(V log V)**.

III. Requerimiento 01

```
def optionThree(analyzer):

    #En primera instancia, se obtiene una estructura que
    diferencia los componentes fuertemente conectados (clusters)

    cluster = controller.getCluster(analyzer)

    #Se obtiene el n mero de clusters dentro de la estructura ya obtenida
    cluster_number = controller.getClusterSize(cluster)

    landing_point_1 = input("Ingrese el nombre del primer landing
____point que desea evaluar: ")
    landing_point_2 = input("Ingrese el nombre del segundo
____landing point que desea evaluar: ")

    #Se evalua si los landing_points ingresados est n conectados en el mismo cluster
    strongly_connected_landing_points = controller.getStronglyConnected(analyzer, cluster,
    landing_point_1, landing_point_2)

    if strongly_connected_landing_points == True:
        strongly_connected_landing_points = "S ."

    else:
        strongly_connected_landing_points = "NO."
```

Como se puede observar, el Requerimiento 1 utiliza las funciones *getCluster()* y *getClusterSize()*, ya que la segunda de estas es una consulta simple, este requerimiento adquiere la complejidad de la primera función únicamente.

Complejidad espacial final: O(V)
Complejidad temporal final: O(V + E)

IV. Requerimiento 02

```
def optionFour(analyzer):

    interconnected_landing_points = controller.getLandingPointConnections(analyzer)
    print(interconnected_landing_points)
```

Este requerimiento únicamente utiliza la función *getLandingPointConnections*, adquiriendo su misma complejidad.

Complejidad espacial final: (O(V log V + V))
Complejidad temporal final: (O(V))

V. Requerimiento 03

```
def optionFive(analyzer):

    country_A = input("Ingrese el pa s desde el que desea encontrar la ruta m nima: ")
    country_B = input("Ingrese el pa s hasta el que desea encontrar la ruta m nima: ")

    dijkstra_info = controller.minimumCountryRoute(analyzer, country_A, country_B)

    minimum_total_distance = dijkstra_info[0]
    minimum_route = dijkstra_info[1]

    #TODO-FORMATEAR PRINT!
    print(minimum_total_distance)
    print(minimum_route)
```

Este requerimiento se basa únicamente en la función *minuminCountryRoute()* por lo que toma su complejidad:

Complejidad espacial final: O(V + E)
Complejidad temporal final: O((V+E) log V)

VI. Requerimiento 04

```
def optionSix(analyzer):

    minimum_spanning_tree = controller.getMST(analyzer)
    print(minimum_spanning_tree) #TODO - FORMATEAR PRINT!
```

Este requerimiento se construye con base en la función *getMST()* por lo que adquiere su misma complejidad temporal:

Complejidad espacial final: O(V)
Complejidad temporal final: O(E log V)

VII. Requerimiento 05

```
def optionSeven(analyzer):

    landing_point = input("Ingrese el nombre del landing point
    respecto al cual quiere determinar los efectos de su fallo: ")

    adjacent_vertices = controller.getAdjacentVertices(analyzer,
    landing_point)

    adjacent_countries = controller.getAdjacentCountries(analyzer, adjacent_vertices,
    landing_point)

    sorted_adjacent_countries = controller.sortAdjacentCountries(adjacent_countries)

    print(sorted_adjacent_countries) #TODO-FORMATEAR PRINT!
```

Este requerimiento a diferencia de los anteriores, requiere de funciones de consulta algo más complejas, además de un ordenamiento final. Al combinar la complejidad de *getAdjacentVertices*, *getAdjacentCountries* y *sortAdjacentCountries* se obtiene que:

Complejidad espacial final: O(3V)
Complejidad temporal final: O(V log V + 2V)

VIII. Conclusión

En conclusión, la alta conectividad de los grafos permite generar rutas de forma inteligente mediante el uso de algoritmos específicos que evitan la iteración bruta que se hace inevitable en otros tipos de estructuras de datos. En el caso de este ejemplo, debido a que existía una gran cantidad de información externa relacionada a cada vértice, fue necesario crear estructuras adicionales para relacionar de forma rápida los vértices con valores particulares. Sin embargo, el hecho de que estas fueran consultas simples no afectó en grandes rasgos el desempeño de las funciones desarrolladas. Así mismo se observó que la teoría de grafos es bastante útil para categorizar y optimizar la información siempre y cuando sea aplicada de forma correcta, ya que la base de todos los requisitos fue un algoritmo aplicable a un grafo. El tiempo de ejecución fue razonable para la cantidad información cargada y el espacio en memoria no estreso en ningún momento a la máquina utilizada, por lo que se puede afirmar que las soluciones planteadas en este documento fueron efectivas y exitosas.

Referencias

[1] R. Sedgewick and K. Wayne, Algorithms, 4th Edition. Addison-Wesley, 2011.