María Castro
Valentina Calderón

Requerimiento 1:

Objetivo: Se desea 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.
View:

- Recibe como parámetro el nombre del landing point 1 y el nombre del landing point 2
- Se revisa si se encuentran en el mapa por landing point names del catálogo, para esto se llama a la función del controller getLandingPointId()
- Si no existen se dice que alguno de los landing point ingresados por parámetro no es válido.
- Si existen, llama la función del controller calcConnectedComponents(). Posteriormente retorna el número total de clústeres presentes en la red e informa si los dos landing points están en el mismo clúster o no.

Controller:

- Llama a la función del modelo getLandingPointId()
- Llama a la función del modelo calcConnectedComponents()

Model:

- getLandingPointId()
    - Se revisa si el landing point ingresado por parámetro se encuentra en el mapa de landing point names. Para esto, se usa un while que recorre el string y verifica si estos hacen parte del mapa por landing points. El recorrido se hace debido a que el usuario únicamente ingresa parte del 'name' del landing point, entonces un .get() diría que el LP no se encuentra.
    - En cada iteración, revisa si el lp ingresado esta en parte del nombre de una llave del mapa
        - Si lo está guarda el id correspondiente y pone notFound como falso para acabar el ciclo
- calcConnectedComponents():
    - Usamos la función de Kosaraju (scc.KosarajuSCC) que crea y retorna la estructura con los componentes conectados.
    - A partir de la estructura, usamos la función del módulo scc connectedComponents() para sacar el número total de clústeres presentes en la red.
    - Por último, usamos la función del módulo scc stronglyConnected() para informar si los dos landing points están en el mismo clúster o no.

Complejidad:

- getLandingPointId → peor caso O(n), n es la cantidad de landingpoints
- calcConnectedComponents → O(V + E)
- Complejidad final: **lineal**, ambas complejidades son lineales

Requerimiento 2:

Objetivo:Se desea encontrar el (los) landing point(s) que sirven como punto de interconexión a más cables en la red.
View:

- Llama a la función pointsInterconnection() del controller
- Itera sobre el/los landinpoints que tiene(n) más puntos de interconexión e imprime en la consola el nombre, país, identificador de cada uno y el total de cables conectados a dichos landing points.
- *Se usa la librería de Folium para generar un mapa que muestra los landing points más conectados (en rojo) y sus conexiones correspondientes a todos los landinpoints que se conectan (en gris) **Req 8*

Controller:

- Llama a la función pointsInterconnection() del modelo

Model:

- Usamos el keySet del mapa por landinpoints para sacar todas las llaves de landingpoints del mapa.
- Itera sobre los landingpoints que hay en el mapa de ladingpoints.
    - Para cada landing point, saca el mapa de cables asociados. El tamaño de este mapa determina el grado del landing point (cuantos cables llegan o salen de él)
    - Si el grado actual es igual mayor que se tiene como mayor, lo agrega a la lista de los landing points más interconectados.
    - Si el grado es que se tiene como mayor, borra los elementos de la lista (inicializan de nuevo con newList()), agrega el landingpoint a esta nueva lista, actualiza el nuevo grado mayor.

Complejidad:

pointInterconnection(): O(n) donde ‘n’ es la cantidad de landing points que hay.

Requerimiento 3:

Objetivo: Se desea 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.

View:

- Se recibe como entrada un país A y un país B
- Se llama la función del controller getCapitalCity() que revisa si existen en el mapa de países del catálogo.
- Se llama a la función del controller minimumDistanceCountries() la cual retorna el path entre el país A y , si existe.
- Itera sobre este path (que es un stack),
    - En cada iteración saca el elemento (edge) en el tope de la pila y para cada uno imprime los dos vértices que conecta y el peso (la distancia de conexión [km] entre cada par consecutivo de landing points).
- Finalmente, se retorna la la distancia total de la ruta
- *se llama a printMapDijkstra(). Esta función usa la librería de folium para crear un mapa mostrando los landing points que forma la ruta mínima entre los dos países*
    - *Las capitales de cada país se muestran de rojo **Req 8*
    - *Los landing points que las conectan se muestran en gris*
      También se crean línea entre puntos adyacentes (no siguen el camino del cable que los conecta si no representa que hay una conexión entre puntos)

Controller:

- Se llama a la función del modelo getCapitalCity()
- Se llama a la función del modelo minimumDistanceCountries()

Model:

- getCapitalCity()
    - Usamos .get() para verificar que el país exista en el mapa de países del catálogo. Si lo encuentra, actualiza la variable a retornar con la capital del país.
- minimumDistanceCountries()
    - Guarda la estructura retornada por el algoritmo del Dijkstra a partir de la capital del primer país.
    - Según esta estructura calcula el camino entre la capital 1 y la capital 2.
    - Retorna el camino entre el país A y B si existe.

Complejidad:

- getCapitalCity() O(1)
- minimumDistanceCountries(), Dijsktra: **O(E log V)** (linearitmico)

Requerimiento 4:

Objetivo: Se requiere identificar la infraestructura crítica para poder garantizar el mantenimiento preventivo del mismo.

View:

- Se llama a la función del controller findGraphMST()
- Se imprime:
    - El número de nodos conectados a la red de expansión mínima
    - El costo total (distancia en [km]) de la red de expansión mínima
    - Presentar la rama más larga (mayor número de arcos entre la raíz y la hoja) que hace parte de la red de expansión mínima
        - Para esto itera sobre el path retornado que es una pila de vértices y en cada iteración imprime el vertices anterior (si lo hay) y el actual

Controller:

- Se llama a la función del modelo findGraphMST()

Model:

- findGraphMST()
    - Usamos la función PrimMst() para calcular la estructura del MST.
    - Usamos la función edgesMST() que retorna una lista con el MST.
    - Usamos la función prim.weightMST() para calcular el peso total del MST.
    - Para encontrar la cantidad de nodos usamos lt.size() sobre la llave ‘mst’ de la estructura
    - Llamamos a la función del modelo createMSTgraph()
    - Llamamos a la función getMSTroots()

- CeateMSTgraph()
    - La función convierte la lista del MST, que contiene los edges, en un grafo.
    - Por cada elemento de la lista, que realmente es un edge, se saca los dos vértices que lo componen y estos se agregan al grafo. Después se crea el arco entre estos dos vértices, con su respectivo peso.

- getMSTroots()

    - Encuentra los nodos que tienen un grado de entrada (indegree) de 0, es decir las raíces del árbol. También se verifica que estos nodos tengan un outdegree mayor a 0, para asegurar que si conectan con otros nodos.
    - Cada nodo con esta propiedad lo añade a una lista.
    - Después de calcular la/las raíces, se llama la función del módulo longestBranch () para calcular y devolver la rama más larga entre el root y alguna hoja del árbol.
    - Se calcula la rama más larga para cada raíz, y se retorna la mayor de todas las calculadas
    -

- longestBranch ()
    - Calcula un BFS desde la raíz que recibe.
    - Itera sobre todos 'visited' del BFS, es decir todos los nodos a los que llega la raíz. Saca la distancia para llegar y si esta es la mayor guarda los datos del vértice y actualiza la distancia mayor.
    - Retorna la distancia máxima, el vértice correspondiente y la estructura calculada a partir del BFS.

Complejidad:

- findGraphMST() →Prim: **O(E log V) (linearitmico)** (complejidad más grande de todas)
- createGraphMST() → O(E) E es la cantidad de edges que hay en el MST
- getMSTroots → O(V), V es la cantidad de nodos en el MST
- longestBranch → BFS: O(V + E), O(V) para ver cual es la rama más grande.

Requerimiento 5:

Objetivo:Se quiere 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; los países afectados son aquellos que cuentan con landing points directamente conectados con el landing point afectado.

View:

- Recibe como entrada el nombre del landing point
- Se llama a la función del controller getLandingPointId() para verificar que el landing point exista.
- Si existe, se llama a la función del controller failureOfLP()
- Se retorna el número de países afectados y una lista de los países afectados.
- *Se crea un mapa usando la librería de folium que muestra el landing point que falló (en rojo) y los países afectados (centrados en la capital) (en gris). Además muestra la distancia entre el lp y la capital. **Req 8*

Controller:

- Se llama a la función del modelo getLandingPointId()
- Se llama a la función del modelo failureOfLP()

Model:

- getLandingPointId()
    - Se revisa si el landing point ingresado por parámetro se encuentra en el mapa de landing point names. Para esto, se usa un while que recorre el

string y verifica si estos hacen parte del mapa por landing points. El recorrido se hace debido a que el usuario únicamente ingresa parte del landing point. para verificar que el landing point exista

- failureOfLP(): Para obtener un el número de países afectados y una lista de los países afectados
    - Se crean dos listas, una para las ciudades y otra para los landing points.
    - Si el landing point que entró como parámetro se encuentra en el catálogo, saca la lista de cables adyacentes al landing point central.
    - Itera sobre la lista de cables que llegan o salen del LP central (ej. 3563)
        - Si se encuentra un cable que contiene 'Land Cable' se guarda en la lista de ciudades ya que estos son los cables que conectan un landing point con la capital
        - Para cada otro cable que no se un cable local (conectan vértices del tipo <Landing Point>-<Cable>, se "arma" el vértice <Landing Point>-<Cable>
            - A partir de estos vértices, se saca su lista de adyacencia del grafo.
            - Se itera sobre los adyacentes y cada uno de estos (que no este todavia en la lista y que sea solo un numero (el id del lp)) se agrega a la lista de lps.
    - Se llama a la función locateLPs (), retorna una lista con los países afectados.
    - Revisa la lista de ciudades (Se espera que solo haya una - capital de donde esta el LP) y añade los países correspondientes si todavía no se encuentra en la lista de países.
    - A partir de las lista de países, se crea una nueva con contiene a estos mismo y a la distancia entre ellos y el landing point que ha fallado
    - Esta lista se ordena a partir de la distancia usando un mergesort.
- locateLPS
    - Determina dónde quedan los LPs adyacentes al que falló
    - Itera sobre cada uno de estos vértices.
        - En cada iteración saca la lista de cables del LP y busca en los cables que sean 'Land Cable' (estos son los conecta un LP con la capital (si la hay))
        - Cuando ya haya encontrado un Land Cable dentro de los cables, determina el país a partir de la capital y usando el mapa de capitals (llave: capital, valor: país)
            - Si el país todavía no se encuentra en la lista, se añade a esta.
        - Cuando ya la haya encontrado la capital se acaba el ciclo
    - Retorna la lista de países afectados

<u>Complejidad:</u>

failureOfLP() → O(c), c es la cantidad de cables que llegan al LP, O(n), n es la cantidad de vértices adyacentes al landing point. O(c * n), n << c.

Mergesort → nLogn, n es la cantidad de países afectados.

locateLps() → O(n) n es la cantidad de vértices adyacentes al landing point, O(c) (peor caso), cable que llegan al vértice adyacente.

- **O(c * n), n siempre va a ser mucho menor que c.**

Pruebas Rendimiento

Requerimiento	Tiempo [ms]	Memoria [Kb]
Requerimiento 1	12188.5327	5213.19
Requerimiento 2	106.475	10.422
Requerimiento 3	2210.10733	3748.11467
Requerimiento 4	7646.88633	6309.52
Requerimiento 5	87.747	2.875