

# Estructuras de Datos y Algoritmos

Jose Luis Tavera - 201821999  
Juan Diego Yepes - 202022391

## Reto IV

### Índice:

<b>1</b>	<b>Introducción</b>	<b>1</b>
<b>2</b>	<b>Construcción del Analizador</b>	<b>2</b>
2.1	Funciones del View . . . . .	2
2.2	Funciones del Controller . . . . .	2
2.3	Funciones del Model . . . . .	3
<b>3</b>	<b>Primer Requerimiento: Vértices Fuertemente Conectados</b>	<b>5</b>
3.1	Funciones del View . . . . .	5
3.2	Funciones del Controller . . . . .	6
3.3	Funciones del Model . . . . .	6
<b>4</b>	<b>Segundo Requerimiento: Vértice con mayor grados</b>	<b>6</b>
4.1	Funciones del View . . . . .	7
4.2	Funciones del Controller . . . . .	7
4.3	Funciones del Model . . . . .	7
<b>5</b>	<b>Tercer Requerimiento: Ruta Mínima entre Dos Países</b>	<b>7</b>
5.1	Funciones del View . . . . .	8
5.2	Funciones del Controller . . . . .	8
5.3	Funciones del Model . . . . .	8
<b>6</b>	<b>Cuarto Requerimiento: Red de Expansión Mínima</b>	<b>9</b>
6.1	Funciones del View . . . . .	9
6.2	Funciones del Controller . . . . .	9
6.3	Funciones del Model . . . . .	9
<b>7</b>	<b>Quinto Requerimiento: Países Afectados por Landing Point</b>	<b>10</b>
7.1	Funciones del View . . . . .	10
7.2	Funciones del Controller . . . . .	11
7.3	Funciones del Model . . . . .	11
<b>8</b>	<b>Sexto Requerimiento: Ancho de Banda por País y Cable</b>	<b>12</b>
8.1	Funciones del View . . . . .	12
8.2	Funciones del Controller . . . . .	12
8.3	Funciones del Model . . . . .	12
<b>9</b>	<b>Séptimo Requerimiento: Ruta mínima entre dos IPs</b>	<b>13</b>
9.1	Funciones del View . . . . .	13
9.2	Funciones del Controller . . . . .	13
9.3	Funciones del Model . . . . .	13

<b>10 Octavo Requerimiento: Mapas de Requerimientos previos</b>	<b>14</b>
10.1 Funciones . . . . .	14
10.2 Resultados Requerimiento 1 . . . . .	17
10.3 Resultados Requerimiento 2 . . . . .	17
10.4 Resultados Requerimiento 3 . . . . .	18
10.5 Resultados Requerimiento 4 . . . . .	18
10.6 Resultados Requerimiento 5 . . . . .	19
<b>11 Tiempos &amp; Espacio de Requerimientos</b>	<b>19</b>
<b>12 Cálculos de Ordenes de Crecimiento</b>	<b>19</b>
12.1 Analizador . . . . .	20
12.2 Primer Requerimiento . . . . .	20
12.3 Segundo Requerimiento . . . . .	20
12.4 Tercer Requerimiento . . . . .	20
12.5 Cuarto Requerimiento . . . . .	21
12.6 Quinto Requerimiento . . . . .	21
12.7 Sexto Requerimiento . . . . .	21
12.8 Séptimo Requerimiento . . . . .	21
12.9 Octavo Requerimiento . . . . .	21
<b>13 Resultados</b>	<b>23</b>

# 1 Introducción

Para dar solución al reto 4, plantemos el siguiente código (ver repositorio):

<https://github.com/EDA2021-1-SEC02-G012/Reto4-G12>

Internet se ha convertido hoy en día en uno de los grandes motores que ha permitido la transformación de la sociedad moderna. Tal ha sido su impacto que se han cambiado la forma en que las personas acceden a la información, la salud, la educación, los trámites gubernamentales, entre otros. Hoy en día tenemos cerca de 4.6 billones de usuarios activos de Internet, esto corresponde a casi un 62% de la población mundial; y en promedio un usuario de Internet está conectado en promedio seis (6) horas y 30 minutos diarios.

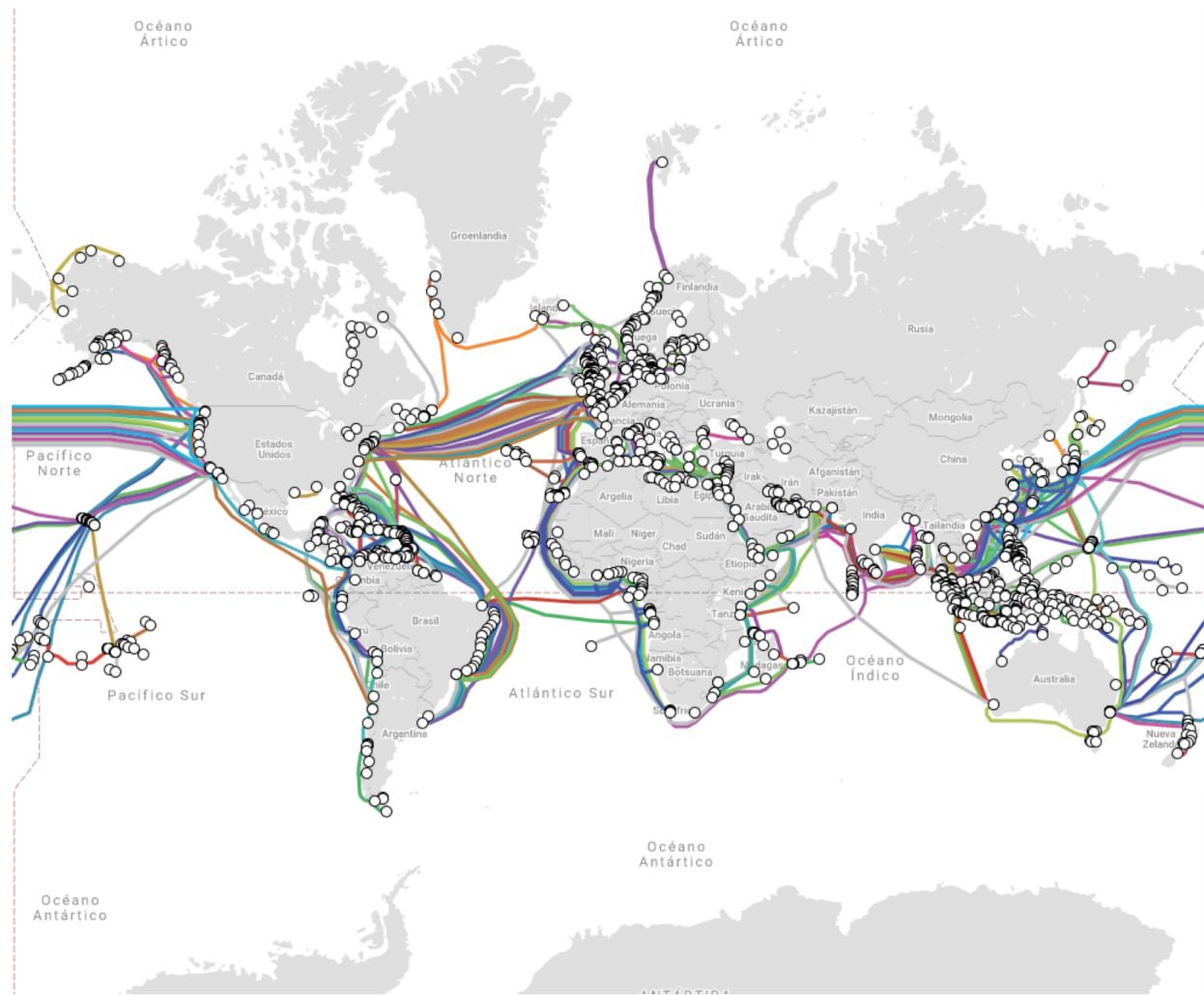


Figure 1: Cables Submarinos

La red de cables submarinos (los que se encuentran instalados en océanos, mares y lagos), es la espina dorsal que sostiene esta gran red de redes que es Internet, ya que permiten la comunicación internacional a una alta velocidad, garantizando el funcionamiento mundial de la banda ancha. Concretamente el 99% de las telecomunicaciones digitales entre continentes y países separados por el mar viajan de esta forma.<sup>2</sup> En su gran mayoría la instalación de estos cables, imitan las rutas utilizadas por los buques de carga que transportan mercancía. En la Figura 1 pueden ver una compilación gráfica de los cables submarinos que actualmente están desplegados a nivel mundial.

Hoy en día hay más de 480 cables que se extienden por más de 1,1 millones de km distribuidos a lo largo y ancho del mundo. Por ejemplo, el SEA-ME-WE3 es un cable de telecomunicaciones submarino óptico que tiene 39.000 kilómetros de longitud y une el Sudeste de Asia con el Oriente Medio y Europa Occidental; actualmente es el cable más largo del mundo.

## 2 Construcción del Analizador

Para construir el analizador, utilizamos en principio un diccionario con las llaves de los mapas, tablas de hash y listas que usaremos después:

```

'landing_points': maptype='PROBING'
'landing_connections': maptype='PROBING'
'landing_point_list': datastructure='ARRAY_LIST'
'countries2': maptype='PROBING'
"country_codes": maptype='PROBING'
'landing_points2': maptype='PROBING'
Analyzer landing_points_map': maptype='PROBING'
'LP_countries': maptype='PROBING'
'cable-LP': maptype='PROBING'
'capitals': maptype='PROBING'
'LP_lat_long': maptype='PROBING'
'Vertex_lat_long': maptype='PROBING'
'connections': graphtype='DIRECTED'

```

### 2.1 Funciones del View

```

1 def optionTwo(analyzer):
2     print("\nCargando información... " + emojis.random_emoji(4))
3     controller.loadLandingPoints(analyzer, landing_points)
4     controller.loadCountries(analyzer, countries)
5     controller.loadConnections(analyzer, connections)
6     print('Se ha cargado la información exitosamente.')

```

### 2.2 Funciones del Controller

```

1 def loadLandingPoints(analyzer, filename):
2     """DS_Store"""
3     landingFile = cf.data_dir + filename
4     input_file = csv.DictReader(open(landingFile, encoding="utf-8"),
5                                 delimiter=",")

```

## 2.3 Funciones del Model

```
1 def cleanLength(connection):
2 """
3     En el caso en el que la distancia sea n.a.
4     se reemplaza por cero
5 """
6     if connection['cable_length'] == 'n.a.':
7         connection['cable_length'] = '10000 km'
8
9 def formatVertex(connection):
10 """
11     Los vértices estarán formateados de la
12     siguiente manera: <LandingPoint>-<Cable>
13 """
14     origin = connection['\ufefforigin'] + '-' + connection['cable_name']
15     destination = connection['destination'] + '-' + connection['cable_name']
16     return origin, destination
```

```

1 def addConnection(analyzer, connection):
2     cleanLength(connection)
3     vertexes = formatVertex(connection)
4     gr.insertVertex(analyzer['connections'], vertexes[0])
5     gr.insertVertex(analyzer['connections'], vertexes[1])
6     weight = connection['cable_length']
7     weight = weight.split(',') [0]
8     if ',' in weight:
9         weight = weight.split(",")
10        weight = float(weight[0])*1000+float(weight[1])
11    else:
12        weight = float(weight)
13    gr.addEdge(
14        analyzer['connections'], vertexes[0], vertexes[1],
15        weight)
16    gr.addEdge(
17        analyzer['connections'], vertexes[1], vertexes[0],
18        weight)

1 def addSantiConnection(analyzer, origin, destination, distance):
2     """
3     Adiciona un arco entre dos estaciones
4     """
5     edge = gr.getEdge(analyzer['connections'], origin, destination)
6     if edge is None:
7         gr.addEdge(analyzer['connections'], origin, destination, distance)
8         gr.addEdge(analyzer['connections'], destination, origin, distance)
9     return analyzer

1 def addGroundConnections(analyzer):
2     """Agrega las conexiones por tierra"""
3     index = 20000
4     prefix = "Capital Connection"
5     listacountries = mp.keySet(analyzer['countries2'])

6     for country in lt.iterator(listacountries):
7         selectcount = mp.get(analyzer['countries2'], country)
8         countri = me.getValue(selectcount)
9         cable = prefix + country
10        origin = str(index) + '-' + cable
11        lp = mp.get(analyzer['landing_points2'], country)
12        gr.insertVertex(analyzer['connections'], origin)
13        boole = lp is not None
14        if boole:
15            mp.put(analyzer['capitals'], country.lower(), origin)
16            lp = me.getValue(lp)
17            lp = lp['cables']
18            for landingPoint in lt.iterator(lp):
19                destination = landingPoint['landing_point_id'] + '-' + cable
20                gr.insertVertex(analyzer['connections'], destination)
21
22                dist = haversine.haversine(
23                    float(countri['CapitalLatitude']),
24                    float(countri['CapitalLongitude']),
25                    float(landingPoint['latitude']),
26                    float(landingPoint['longitude'])))
27
28                gr.addEdge(analyzer['connections'], destination, origin, dist)
29                gr.addEdge(analyzer['connections'], origin, destination, dist)
30                addConnectionToLandingMapVer2(origin, cable, analyzer)
31                addConnectionToLandingMapVer2(destination, cable, analyzer)
32
33        index += 1
34
35    else:
36        closestLP = None
37        minDistance = 100000000
38        for landingPoint in lt.iterator(analyzer['landing_point_list']):
39            distance = haversine.haversine(

```

```

40         float(countri['CapitalLatitude']),
41         float(countri['CapitalLongitude']),
42         float(landingPoint['latitude']),
43         float(landingPoint['longitude']))
44     if distance < minDistance:
45         closestLP = landingPoint
46         minDistance = distance
47     destination = closestLP['landing_point_id']+'-'+cable
48     gr.insertVertex(analyzer['connections'], destination)
49     gr.addEdge(
50         analyzer['connections'], destination, origin, distance)
51     gr.addEdge(
52         analyzer['connections'], origin, destination, distance)
53     addConnectionToLandingMapVer2(origin, cable, analyzer)
54     addConnectionToLandingMapVer2(destination, cable, analyzer)
55     index += 1

1 def relateSameLandings(analyzer):
2     lststops = mp.keySet(analyzer['landing_connections'])
3     for key in lt.iterator(lststops):
4         gr.insertVertex(analyzer['connections'], key)
5         lstroutes = mp.get(
6             analyzer['landing_connections'], key)['value']['cables']
7         prevrout = None
8         for route in lt.iterator(lstroutes):
9             route = key + '-' + route
10            if prevrout is not None:
11                addSantiConnection(analyzer, prevrout, route, 100)
12                addSantiConnection(analyzer, route, prevrout, 100)
13                addSantiConnection(analyzer, key, route, 100)
14                addSantiConnection(analyzer, key, prevrout, 100)
15                addSantiConnection(analyzer, prevrout, key, 100)
16                addSantiConnection(analyzer, route, key, 100)
17            prevrout = route

```

### 3 Primer Requerimiento: Vértices Fuertemente Conectados

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. Para dar respuesta a este requerimiento el equipo de desarrollo debe recibir como entrada la siguiente información:

- Nombre del landing point 1
- Nombre del landing point 2

Y como respuesta debe presentar en consola la siguiente información:

- Número total de clústeres presentes en la red
- Informar si los dos landing points están en el mismo clúster o no.

#### 3.1 Funciones del View

```

1 def optionThree(analyzer):
2     '',
3     VertexA, VertexB
4     Encontrar cl steres: Kosaraju,
5     decir si dos v rtices est n conectados
6     '',
7     graph = analyzer['connections']
8     vertexA = input('Ingrese el v rtice de origen: ')
9     vertexA = vertexA.lower()
10    vertexA_cod = controller.searchCountry(vertexA, analyzer)

```

```

11  vertexB = input('Ingrese el vertice destino: ')
12  vertexB = vertexB.lower()
13  vertexB_cod = controller.searchCountry(vertexB, analyzer)
14  Req1 = (controller.req1(graph, vertexA_cod, vertexB_cod))
15  if Req1[0] is True:
16      print('\n')
17      print('Los vertices se encuentran fuertemente conectados')
18  else:
19      print('Los vertices no se encuentran fuertemente conectados')
20  print('El numero de total de clusters en la red es de: ' + str(Req1[1]))

```

## 3.2 Funciones del Controller

```

1 def searchCountry(name, analyzer):
2     return model.searchCountry(name, analyzer)
3
4 def req1(graph, vertexA, vertexB):
5     if (vertexA is not None) and (vertexB is not None):
6         tree = model.Kosaraju(graph)
7         path = model.arestronglyConnected(tree, vertexA, vertexB)
8         clustersAB = model.sccCount(graph, tree, vertexA)[‘components’]
9         return (path, clustersAB)
10    else:
11        return "Error en los vertices"

```

## 3.3 Funciones del Model

```

1 def searchCountry(name, analyzer):
2     mapa = analyzer[‘landing_points’]
3     keyvalue = mp.get(mapa, name)
4     if keyvalue is not None:
5         code = me.getValue(keyvalue)
6         return code
7     else:
8         return None
9
10 def Kosaraju(graph):
11     return scc.KosarajuSCC(graph)
12
13 def arestronglyConnected(s, vertexA, vertexB):
14     return scc.stronglyConnected(s, vertexA, vertexB)
15
16 def sccCount(graph, scc2, vert):
17     return scc.sccCount(graph, scc2, vert)

```

## 4 Segundo Requerimiento: Vértice con mayor grados

Se desea encontrar el (los) landing point(s) que sirven como punto de interconexión a más cables en la red. Para dar respuesta a este requerimiento el equipo de desarrollo no necesita ninguna entrada, y como respuesta debe presentar en consola la siguiente información:

- Lista de landing points (nombre, país, identificador).
- Total, de cables conectados a dichos landing points.

## 4.1 Funciones del View

```
1 def optionFour(analyzer):
2     '',
3     Landing point que sirve de interconexión
4     m s arcos: TAD graph: Encontrar vértice con
5     mayor grado
6     Puede haber m s de uno
7     '',
8     definitiva = controller.req2(analyzer)
9     PrintDefinitiva(analyzer, definitiva[0], definitiva[1])
10    controller.graphicateReq2(analyzer, definitiva)
```

## 4.2 Funciones del Controller

```
1 def req2(analyzer):
2     return model.getCriticalVertex(analyzer)
```

## 4.3 Funciones del Model

```
1 def getCriticalVertex(analyzer):
2     mapa = analyzer['landing_connections']
3     keys = mp.keySet(mapa)
4     mayor = -1
5     key = []
6     for i in lt.iterator(keys):
7         if int(i) < 20000:
8             pair = mp.get(mapa, i)
9             value = me.getValue(pair)
10            size = lt.size(value['cables'])
11            if size >= mayor:
12                mayor = size
13                key.append(i)
14
15    definitiva = []
16
17    for k in key:
18        pair = mp.get(mapa, k)
19        value = me.getValue(pair)
20        size = lt.size(value['cables'])
21        if size == mayor:
22            definitiva.append(k)
23
24    return definitiva, mayor
```

## 5 Tercer Requerimiento: Ruta Mínima entre Dos Países

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. Para dar respuesta a este requerimiento el equipo de desarrollo debe recibir como entrada la siguiente información:

- País A
- País B

Y como respuesta debe presentar en consola la siguiente información:

- Ruta (incluir la distancia de conexión [km] entre cada par consecutivo de landing points)
- Distancia total de la ruta

Para calcular la distancia entre dos ubicaciones geográficas que tienen latitud y longitud se sugiere la utilización de la Fórmula de Distancia Haversine.

## 5.1 Funciones del View

```
1 def optionFive(analyzer):
2     # TODO Poner capital connection vertex
3     """VertexA, VertexB
4     Ruta m nima en distancia,
5     F rmula Haversine con una librera"""
6     print(
7         "\nADVERTENCIA: La distancia se encontrar entre las capitales de",
8         "los pa ses que seleccione." + emojis.random_emoji(1))
9     paisA = input('Ingrese el pa s origen: ')
10    paisA = paisA.lower()
11    vertexA = controller.searchVertexCountry(paisA, analyzer)
12    paisB = input('Ingrese el pa s destino: ')
13    paisB = paisB.lower()
14    vertexB = controller.searchVertexCountry(paisB, analyzer)
15    ruta = controller.req3(analyzer, vertexA, vertexB)
16    PrintRutaMinima(ruta)
17    controller.graphicateReq3(analyzer, ruta[0])
```

## 5.2 Funciones del Controller

```
1 def searchVertexCountry(pais, analyzer):
2     return model.searchVertexCountry(pais, analyzer)

1 def req3(analyzer, vertexA, vertexB):
2     if (vertexA is not None) and (vertexB is not None):
3         caminominimo = model.DijkstraAlgo(analyzer['connections'], vertexA)
4         path = model.findDistTo(caminominimo, vertexB)
5         ruta = model.createRoute(path)
6         return ruta
7     else:
8         return "Error en los v rtices"
```

## 5.3 Funciones del Model

```
1 def searchVertexCountry(pais, analyzer):
2     mapa = analyzer['capitals']
3     keyvalue = mp.get(mapa, pais)
4     if keyvalue is not None:
5         vertex = me.getValue(keyvalue)
6         return vertex
7     else:
8         return None

1 def DijkstraAlgo(graph, vertexA):
2     return dijsktra.Dijkstra(graph, vertexA)

1 def DijkstraAlgo(graph, vertexA):
2     return dijsktra.Dijkstra(graph, vertexA)

1 def findDistTo(caminominimo, vertexB):
2     return dijsktra.pathTo(caminominimo, vertexB)

1 def createRoute(path):
2     vertices = []
3     distancia = 0
4     vertices.append(['PARADA', 'DISTANCIA'])
5     vertices.append(['', ''])
6     for vertex in lt.iterator(path):
7         vertexA = vertex['vertexA']
8         weight = vertex['weight']
9         distancia += float(weight)
```

```

10     vertices.append([vertexA, weight])
11
12     return (vertices, distancia)

```

## 6 Cuarto Requerimiento: Red de Expansión Mínima

Se requiere 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. Como respuesta debe presentar en consola la siguiente información:

- 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

### 6.1 Funciones del View

```

1 def optionSix(analyzer):
2     """Red de expansión mínima: MST"""
3     req4 = controller.req4(analyzer['connections'])
4     PrintREQ4(req4)
5     controller.graphicateReq4(analyzer, req4[2][2])

```

### 6.2 Funciones del Controller

```

1 def req4(graph):
2     mst = model.findMST(graph)
3     dist_to = mst['distTo']
4     edge_to = mst['edgeTo']
5     no_nodos_conectados = dist_to['size']
6     distance = model.get_total_distance(dist_to)
7     longest_branch = model.doBFS(edge_to)
8
9     return no_nodos_conectados, distance, longest_branch

```

### 6.3 Funciones del Model

```

1 def findMST(graph):
2     mst = prim.PrimMST(graph)
3     return mst
4
5 def get_total_distance(hashtable):
6     keys = mp.keySet(hashtable)
7     dist = 0
8     for i in lt.iterator(keys):
9         dist += float(mp.get(hashtable, i)['value'])
10
11 def doBFS(edgeTo):
12     minigraph = gr.newGraph(
13         datastructure='ADJ_LIST',
14         directed=True,
15         size=1000,
16         comparefunction=cmplandingpoints)
17     keys = mp.keySet(edgeTo)
18     firstVertex = lt.getElement(keys, 1)

```

```

9     for i in lt.iterator(keys):
10        vertexA = i
11        vertexB = mp.get(edgeTo, i)[‘value’][‘vertexA’]
12        gr.insertVertex(minigraph, vertexA)
13        gr.insertVertex(minigraph, vertexB)
14
15    for i in lt.iterator(keys):
16        vertexA = i
17        vertexB = mp.get(edgeTo, i)[‘value’][‘vertexA’]
18        weight = mp.get(edgeTo, i)[‘value’][‘weight’]
19        gr.addEdge(minigraph, vertexA, vertexB, weight)
20        gr.addEdge(minigraph, vertexB, vertexA, weight)
21
22    distNsize = {}
23    vertexes = gr.vertices(minigraph)
24    mayor = 0
25
26    recorrido = bfs.BreadhtFisrtSearch(minigraph, firstVertex)
27    for vertexB in lt.iterator(vertexes):
28        dist = bfs.pathTo(recorrido, vertexB)
29        if dist is not None:
30            size = lt.size(dist)
31            distNsize[size] = dist
32            if size > mayor:
33                mayor = size
34
35    for i in distNsize.keys():
36        if i == mayor:
37            caminomaslargo = distNsize[i]
38
39    return mayor, caminomaslargo, minigraph

```

## 7 Quinto Requerimiento: Países Afectados por Landing Point

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. Para dar respuesta a este requerimiento el equipo de desarrollo debe recibir como entrada la siguiente información:

- Nombre del landing point

Y como respuesta debe presentar en consola la siguiente información:

- Número de países afectados
- Lista de los países afectados (la lista debe darse por distancia en km descendente)

### 7.1 Funciones del View

```

1 def optionSeven(analyzer):
2     """Impacto que tendrá el fallo de un LP"""
3     LP = input('Ingrese el landing point que fall : ')
4     landingPoint = controller.searchCountry(LP.lower(), analyzer)
5     paises = controller.req5(analyzer, landingPoint)
6     sorted_paises = controller.SortCountries(analyzer, paises, LP)
7     PaisesAfectados(sorted_paises)
8     controller.graphicateReq5(analyzer, paises, LP)

```

## 7.2 Funciones del Controller

```
1 def searchCountry(name, analyzer):
2     return model.searchCountry(name, analyzer)

1 def req5(analyzer, landing_point):
2     lista = model.findCountriesFromAdjacents(
3         analyzer['connections'], landing_point)
4     countries = model.primleisi(analyzer, lista)
5     return countries

1 def SortCountries(analyzer, paises, LP):
2     return model.SortCountries(analyzer, paises, LP)
```

## 7.3 Funciones del Model

```
1 def searchCountry(name, analyzer):
2     mapa = analyzer['landing_points']
3     keyvalue = mp.get(mapa, name)
4     if keyvalue is not None:
5         code = me.getValue(keyvalue)
6         return code
7     else:
8         return None

1 def findCountriesFromAdjacents(graph, landingPoint):
2     list_adjacents = gr.adjacents(graph, landingPoint)
3     countries = []
4     for vertex in lt.iterator(list_adjacents):
5         adjacents_vertex = gr.adjacents(graph, vertex)
6         for adjacent in lt.iterator(adjacents_vertex):
7             countries.append(adjacent)
8             if landingPoint in adjacent:
9                 otra_variable = gr.adjacents(graph, adjacent)
10                for otro_adjacente in lt.iterator(otra_variable):
11                    countries.append(otro_adjacente)
12
13 return countries

1 def primleisi(analyzer, listaDario):
2     country_base = analyzer['LP_countries']
3     lista = []
4     for i in listaDario:
5         vertex = i.split('-')
6         vertex = vertex[0]
7         if float(vertex) < 20000 and vertex is not None:
8             country = mp.get(country_base, vertex)['value']
9             lista.append(country)
10        else:
11            country = i.split('-')
12
13 lista = list(set(lista))
14 return lista

1 def SortCountries(analyzer, paises, LP):
2     countries = analyzer['countries2']
3     sorted_paises = {}
4     landingPoint_id = mp.get(analyzer['landing_points'], LP.lower())['value']
5     coord = mp.get(analyzer['LP_lat_long'], landingPoint_id)['value']
6     for countri1 in paises:
7         countri = mp.get(countries, countri1)['value']
8         dist = haversine.haversine(
9             float(countri['CapitalLatitude']),
10            float(countri['CapitalLongitude']),
11            float(coord[0]),
```

```

12         float(coord[1]))
13
14     sorted_paises[country1] = dist
15
16     return sorted_paises

```

## 8 Sexto Requerimiento: Ancho de Banda por País y Cable

Se quiere 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. Para tal fin debe calcular el ancho de banda máximo ofrecido por el landing point en cada país de acuerdo con el ancho de banda del cable conectado. Para dar respuesta a este requerimiento el equipo de desarrollo debe recibir como entrada la siguiente información:

- Nombre del país
- Nombre del cable

Y como respuesta debe presentar en consola la siguiente información:

- Lista de los países conectados al cable con el ancho de banda máximo que puede garantizarse

### 8.1 Funciones del View

```

1 def optionEight(analyzer):
2     pais = input('Ingrese el nombre del país: ')
3     cable = input('Ingrese el nombre del cable: ')
4     anchos_banda = (controller.req6(analyzer, pais, cable))
5     PrintAnchodeBanda(anchos_banda)

```

### 8.2 Funciones del Controller

```

1 def req6(analyzer, pais, cable):
2     return model.findIfCableInCountry(analyzer, pais, cable)

```

### 8.3 Funciones del Model

```

1 def findIfCableInCountry(analyzer, pais, cable):
2     mapa = analyzer['cable-LP']
3     lps = mp.get(mapa, cable)
4     alejandra = []
5     for i in lt.iterator(lps['value']['landing_points']):
6         alejandra.append(i['\ufefforigin'])
7     listaNORMAL = primeleisi(analyzer, alejandra)
8     return elazotanalgas3000(analyzer, listaNORMAL, cable, pais)
9
10
11 def elazotanalgas3000(analyzer, listaNORMAL, cable, pais2):
12     countries_map = analyzer['countries2']
13     cable_info_map = analyzer['cable-LP']
14     dick = {}
15     for pais in listaNORMAL:
16         if pais.lower() != pais2.lower():
17             no_usuarios = int(
18                 mp.get(countries_map, pais)['value']['Internet users'])
19             info1 = mp.get(cable_info_map, cable)['value']['landing_points']
20             capacidad = float(lt.firstElement(info1)['capacityTBPS'])
21             dick[pais] = ((capacidad*8388608)/no_usuarios)
22
23     return dick

```

## 9 Séptimo Requerimiento: Ruta mínima entre dos IPs

Se desea encontrar la ruta mínima en número de saltos para enviar información entre dos direcciones IP dadas. Teniendo en cuenta que para cada IP se puede obtener la información de en qué país está ubicada (utilizando el API ip-api), ustedes deberán encontrar el landing point más cercano para cada dirección IP. Para dar respuesta a este requerimiento el equipo de desarrollo debe recibir como entrada la siguiente información:

- Dirección IP1
- Dirección IP2

Y como respuesta debe presentar en consola la siguiente información:

- Ruta
- Número de saltos de la ruta

### 9.1 Funciones del View

```
1 def optionNine(analyzer):  
2     '''Dada la IP encontrar el país y luego encontrar ruta mínima'''  
3     ip_1 = input('Ingrese la IP de or gen: ')  
4     location_1 = controller.getLocation(ip_1, analyzer)  
5     vertexA = controller.searchVertexCountry(location_1, analyzer)  
6     ip_2 = input('Ingrese la IP de destino: ')  
7     location_2 = controller.getLocation(ip_2, analyzer)  
8     vertexB = controller.searchVertexCountry(location_2, analyzer)  
9     ruta = (controller.req3(analyzer, vertexA, vertexB))  
10    PrintRutaMinima(ruta)
```

### 9.2 Funciones del Controller

```
1 def getLocation(ip, analyzer):  
2     return model.getLocation(ip, analyzer)  
  
1 def searchVertexCountry(pais, analyzer):  
2     return model.searchVertexCountry(pais, analyzer)
```

### 9.3 Funciones del Model

```
1 def getLocation(ip, analyzer):  
2     loc = get('https://ipapi.co/{ip}/json/'.format(ip=ip))  
3     info = loc.json()  
4     code = info['country_code']  
5     country = mp.get(analyzer['country_codes'], code)  
6     country = me.getValue(country)  
7     country = country.lower()  
8     return country  
  
1 def DijkstraAlgo(graph, vertexA):  
2     return dijsktra.Dijkstra(graph, vertexA)  
  
1 def DijkstraAlgo(graph, vertexA):  
2     return dijsktra.Dijkstra(graph, vertexA)  
  
1 def findDistTo(caminominimo, vertexB):  
2     return dijsktra.pathTo(caminominimo, vertexB)
```

```

1 def createRoute(path):
2     vertices = []
3     distancia = 0
4     vertices.append(['PARADA', 'DISTANCIA'])
5     vertices.append(['', ''])
6     for vertex in lt.iterator(path):
7         vertexA = vertex['vertexA']
8         weight = vertex['weight']
9         distancia += float(weight)
10        vertices.append([vertexA, weight])
11
12    return (vertices, distancia)

1 def searchVertexCountry(pais, analyzer):
2     mapa = analyzer['capitals']
3     keyvalue = mp.get(mapa, pais)
4     if keyvalue is not None:
5         vertex = me.getValue(keyvalue)
6         return vertex
7     else:
8         return None

```

## 10 Octavo Requerimiento: Mapas de Requerimientos previos

Se otorgará una bonificación a los equipos de trabajo que grafiquen en un mapa los resultados de cada uno de los requerimientos anteriormente enunciados.

### 10.1 Funciones

```

1 def graphicateReq1(analyzer, route):
2     map1 = folium.Map()
3     route.pop(0)
4     route.pop(0)
5     rutas = []
6
7     for i in range(len(route)):
8         landinpoint = route[i][0]
9         if len(landinpoint) > 5:
10             split = landinpoint.split('-')
11             split = split[0]
12             if float(split) < 20000:
13                 verticei = mp.get(analyzer['LP_lat_long'], split)['value']
14                 folium.Marker(verticei).add_to(map1)
15                 rutas.append(verticei)
16             else:
17                 split2 = landinpoint.split("Capital Connection ")
18                 split2 = split2[1]
19                 countri = mp.get(analyzer['countries2'], split2)['value']
20                 verticei = [float(countri['CapitalLatitude']), float(
21                     countri['CapitalLongitude'])]
22                 folium.Marker(verticei).add_to(map1)
23                 rutas.append(verticei)
24
25     folium.PolyLine(rutas).add_to(map1)
26
27     direction = cf.data_dir + 'MapREQ1.html'
28     map1.save(direction)

1 def graphicateReq2(analyzer, definitiva):
2     map2 = folium.Map()
3     vertice = mp.get(analyzer['LP_lat_long'], definitiva[0][0])['value']

```

```

4     folium.Marker(vertice).add_to(map2)
5
6     adyacentes = gr.adjacents(analyzer['connections'], definitiva[0][0])
7     gral_list = []
8
9     for i in lt.iterator(adyacentes):
10        adj = gr.adjacents(analyzer['connections'], i)
11        for k in lt.iterator(adj):
12            if definitiva[0][0] not in k:
13                gral_list.append(k)
14
15     for i in gral_list:
16         split = i.split('-')
17         split = split[0]
18         if float(split) < 20000:
19             verticei = mp.get(analyzer['LP_lat_long'], split)['value']
20             folium.Marker(verticei).add_to(map2)
21             route = [vertice, verticei]
22             folium.PolyLine(route).add_to(map2)
23         else:
24             split2 = i.split("Capital Connection ")
25             split2 = split2[1]
26             countri = mp.get(analyzer['countries2'], split2)['value']
27             verticei = [float(countri['CapitalLatitude']), float(
28                         countri['CapitalLongitude'])]
29
30             folium.Marker(verticei).add_to(map2)
31             route = [vertice, verticei]
32             folium.PolyLine(route).add_to(map2)
33
34     direction = cf.data_dir + 'MapREQ2.html'
35     map2.save(direction)
36
37
38 def graphicateReq3(analyzer, route):
39     map3 = folium.Map()
40     route.pop(0)
41     route.pop(0)
42     rutas = []
43
44     for i in range(len(route)):
45         landinpoint = route[i][0]
46         if len(landinpoint) > 5:
47             split = landinpoint.split('-')
48             split = split[0]
49             if float(split) < 20000:
50                 verticei = mp.get(analyzer['LP_lat_long'], split)['value']
51                 folium.Marker(verticei).add_to(map3)
52                 rutas.append(verticei)
53             else:
54                 split2 = landinpoint.split("Capital Connection ")
55                 split2 = split2[1]
56                 countri = mp.get(analyzer['countries2'], split2)['value']
57                 verticei = [float(countri['CapitalLatitude']), float(
58                         countri['CapitalLongitude'])]
59                 folium.Marker(verticei).add_to(map3)
60                 rutas.append(verticei)
61
62     folium.PolyLine(rutas).add_to(map3)
63
64     direction = cf.data_dir + 'MapREQ3.html'
65     map3.save(direction)
66
67
68 def graphicateReq4(analyzer, minigraph):
69     map4 = folium.Map()
70     vertexes = gr.vertices(minigraph)
71     edges = gr.edges(minigraph)
72
73

```

```

6   for i in lt.iterator(vertexes):
7     split = i.split(',')
8     split = split[0]
9     if float(split) < 20000:
10       vertice = mp.get(analyzer['LP_lat_long'], split)['value']
11       folium.Marker(vertice).add_to(map4)
12     else:
13       split2 = i.split("Capital Connection ")
14       split2 = split2[1]
15       countri = mp.get(analyzer['countries2'], split2)['value']
16       verticei = [float(countri['CapitalLatitude']), float(
17         countri['CapitalLongitude'])]
18       folium.Marker(verticei).add_to(map4)
19
20   for i in lt.iterator(edges):
21     vertexA = i['vertexA']
22     vertexB = i['vertexB']
23     splitA = vertexA.split(',')
24     splitA = splitA[0]
25     splitB = vertexB.split(',')
26     splitB = splitB[0]
27     route = []
28
29     if float(splitA) < 20000:
30       verticei = mp.get(analyzer['LP_lat_long'], splitA)['value']
31       route = [verticei]
32     else:
33       split2 = vertexA.split("Capital Connection ")
34       split2 = split2[1]
35       countri = mp.get(analyzer['countries2'], split2)['value']
36       verticei = [float(countri['CapitalLatitude']), float(
37         countri['CapitalLongitude'])]
38       route = [verticei]
39
40     if float(splitB) < 20000:
41       verticei = mp.get(analyzer['LP_lat_long'], splitB)['value']
42       route.append(verticei)
43     else:
44       split2 = vertexB.split("Capital Connection ")
45       split2 = split2[1]
46       countri = mp.get(analyzer['countries2'], split2)['value']
47       verticei = [float(countri['CapitalLatitude']), float(
48         countri['CapitalLongitude'])]
49       route.append(verticei)
50
51   folium.PolyLine(route).add_to(map4)
52
53 direction = cf.data_dir + 'MapREQ4.html'
54 map4.save(direction)
55
56
1 def graphicateReq5(analyzer, paises, LP):
2   map5 = folium.Map()
3   landingPoint_id = mp.get(analyzer['landing_points'], LP.lower())['value']
4   coord = mp.get(analyzer['LP_lat_long'], landingPoint_id)['value']
5   folium.Marker(coord).add_to(map5)
6
7   for pais in paises:
8     countri = mp.get(analyzer['countries2'], pais)['value']
9     countrycords = [float(countri['CapitalLatitude']), float(
10       countri['CapitalLongitude'])]
11     folium.Marker(countrycords).add_to(map5)
12     route = [coord, countrycords]
13     folium.PolyLine(route).add_to(map5)
14
15 direction = cf.data_dir + 'MapREQ5.html'
16 map5.save(direction)

```

## 10.2 Resultados Requerimiento 1

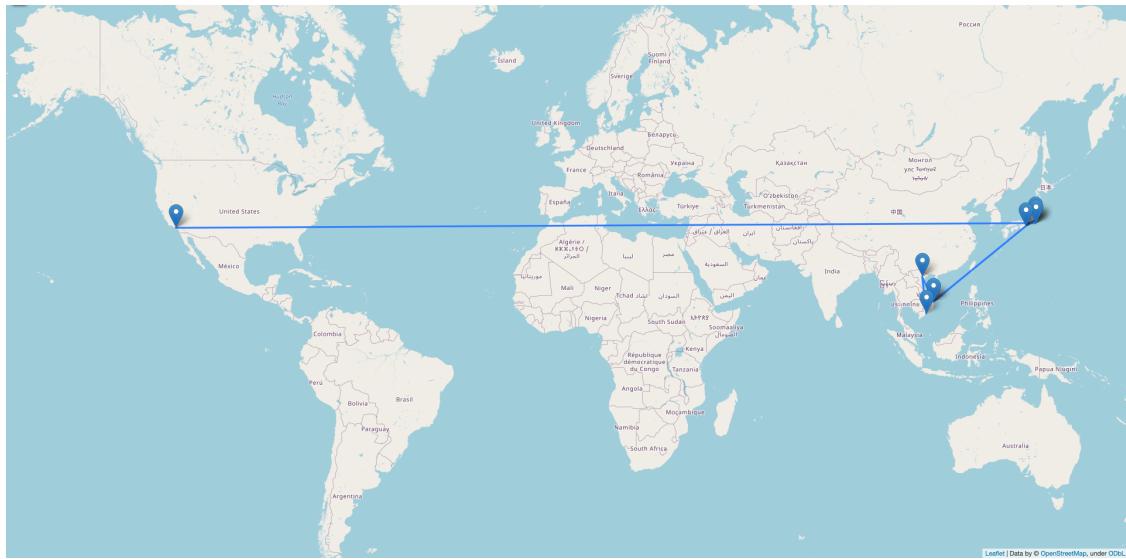


Figure 2: Vértices Fuertemente Conectados - Vung Tau y Redondo Beach

## 10.3 Resultados Requerimiento 2

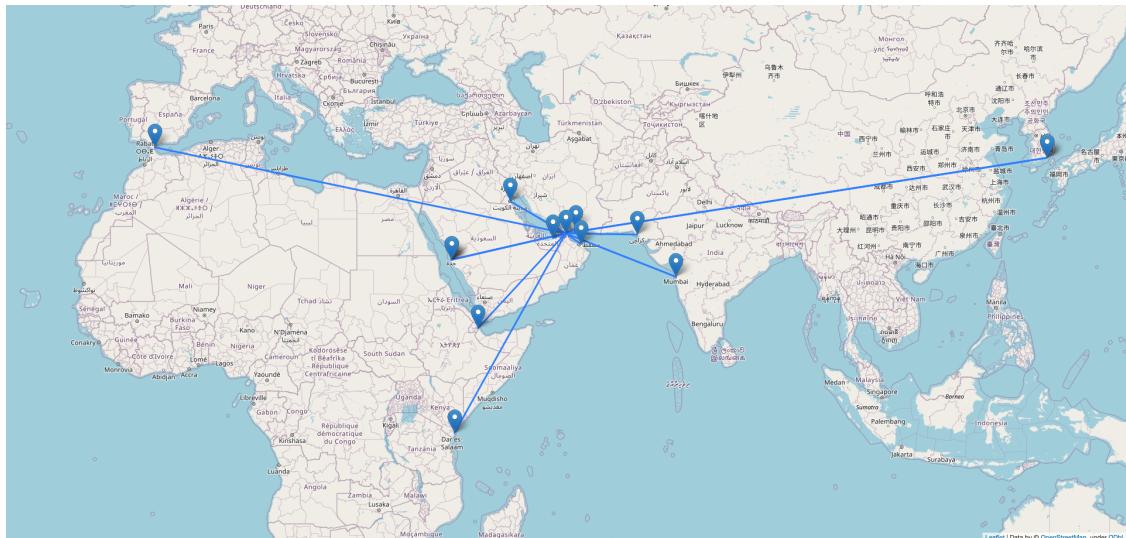


Figure 3: Cables Submarinos

## 10.4 Resultados Requerimiento 3

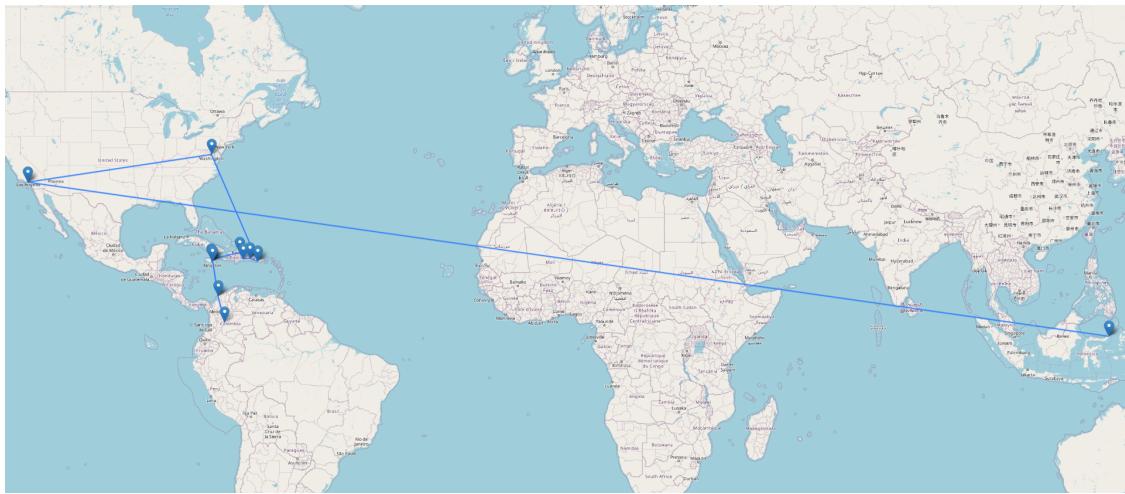


Figure 4: Ruta Mínima entre Países - Colombia e Indonesia

## 10.5 Resultados Requerimiento 4

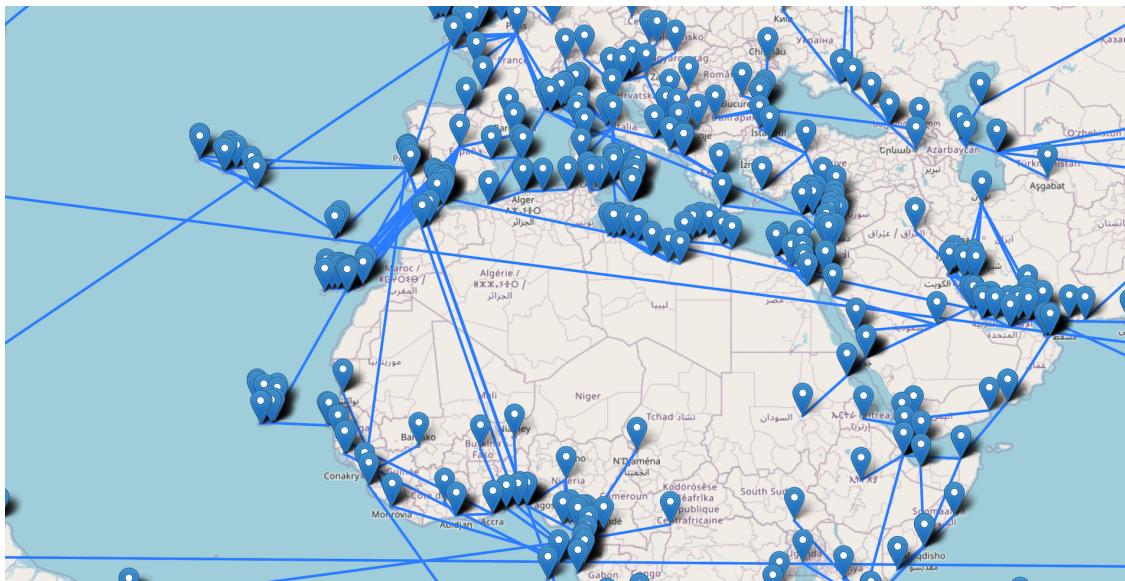


Figure 5: Red de Expansión Mínima

## 10.6 Resultados Requerimiento 5

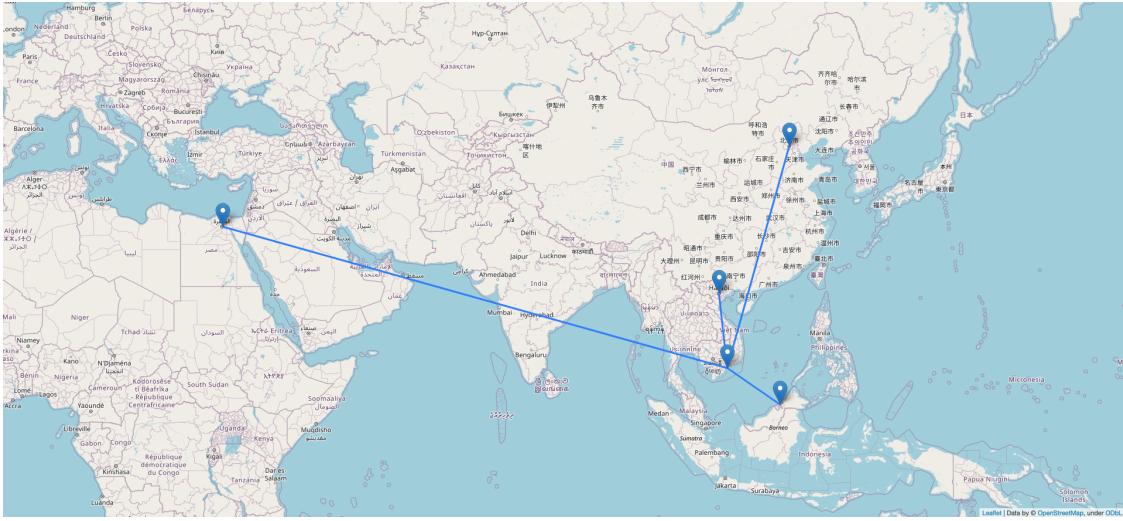


Figure 6: Países Afectadas - Falla en Vung Tau

## 11 Tiempos & Espacio de Requerimientos

Req	Tiempo (ms)	Espacio (kb)	Tiempo graficación (ms)	Espacio graficación (kb)
0	3264.117	22277.873	N/A	N/A
1	8328.088	137.137	3475.6033	155.607
2	67.198	4.632	54.031	185.249
3	9965.468	22.778	39.253	147.220
4	14361.574	13832.290	19370.883	84930.475
5	18.185	1.25	53.881	116.310
6	1.101	2.140	N/A	N/A
7	6108.192	20.859	N/A	N/A

**Tabla 1:** Análisis de Tiempo y Espacio por requerimiento.

El literal 0 hace referencia a la carga de datos.

## 12 Cálculos de Órdenes de Crecimiento

En el siguiente apartado analizaremos los cálculos de los ordenes de crecimiento de cada una de las funciones. En primera instancia, es necesaria definir distintos "tamaños de datos" para cada archivo y filtración usada en los análisis, ya que sería negligente referirse a un solo tamaño "N" de datos. Por lo tanto, definiremos las siguientes variables para su mejor y peor caso:

- **CTS:** Indica el tamaño de entradas dentro del archivo .csv correspondiente a "Countries".
  - **LP:** Indica el tamaño de entradas dentro del archivo .csv correspondiente a "Landing Points".
  - **CNN:** Indica el tamaño de entradas dentro del archivo .csv correspondiente a "Connections".

Como los anteriores tienen un número constante de entradas, no tienen peor ni mejor caso. Por otro lado, definiremos un tamaño de datos de filtración por rango para cualquiera de las características musicales, y un subtamaño correspondiente a una segunda filtración que nos servirán para analizar los ordenes de crecimiento de los requerimientos:

- **V:** Es el número de vértices en el grafo y es constante por lo que no tiene mejor, ni peor caso.
- **E:** Es el número de arcos en grafo y es constante por lo que no tiene mejor, ni peor caso.
- **$V'$ :** Es el tamaño promedio de la lista de adyacentes de un vértice cualquiera. Su peor caso es  $V'_w$  y su mejor caso lo indicaremos con  $V'_B$ .
- **C:** Es el número promedio de cables a los que está conectado cada landing point, que en su peor caso es igual a  $C_W$  y en su mejor caso diremos que es igual a  $C_B$ .
- **DJ:** Es el número de vértices promedio obtenido después de hacer el algoritmo de dijkstra, que en su peor caso es igual a  $DJ_W$  y en su mejor caso lo señalaremos con  $DJ_B$ .

## 12.1 Analizador

Como para la construcción de todas las estructuras de datos, solo se iteraron los tres archivos una vez, entonces, asumiendo que no hacen rehash los mapas, podemos decir que su orden de crecimiento es igual a:

$$A = CTS + LP + CNN$$

Que no tienen peor, ni mejor caso ya que los archivos .csv manejan entradas constantes.

## 12.2 Primer Requerimiento

Para el primer requerimiento necesitábamos confirmar si dos vértices se encontraban fuertemente conectados. Por lo cual, seguimos los siguientes pasos:

1. Implementamos el algoritmo de Kosaraju, el cual realiza dos veces un recorrido sobre el grafo en  $V + E$
2. Por medio de la función de `areStronglyConnected`, analizamos si dos vértices están fuertemente conectados. La librería analiza los clusters para cada uno de los vértices, en el caso de que sean iguales y pertenezcan al mismo clúster se retorna `True`, de lo contrario se retorna `False`. Al tratarse de mapas hechos por el algoritmo de Kosaraju anterior, estas operaciones se realizan en  $O(1)$

En total, su orden de crecimiento sería igual a:

$$Req1 = V + E$$

## 12.3 Segundo Requerimiento

En el segundo requerimiento, necesitábamos hallar el vértice con mayor número de grados. Para lograr lo anterior iteramos el mapa de `landing_connections`, que tenía como llave los landing points y como valor los cables conectados al landing point respectivo. Por lo tanto, el costo de esta iteración es  $LP$ . En total, su orden de crecimiento sería igual a:

$$Req2 = LP$$

## 12.4 Tercer Requerimiento

Para hallar la ruta mínima entre dos países era necesario utilizar el algoritmo de dijkstra que encuentra los caminos mínimos para un vértice (en nuestro caso el vértice de origen). El orden de crecimiento correspondiente a este es igual a  $E + V \log(V)$ . Ahora bien, para hallar el camino entre el vértice de origen y el de destino usamos la función de `pathTo` que debe iterar los mapas creados por dijkstra que en su peor caso tiene un orden de crecimiento igual a  $V$ . En total, su orden de crecimiento sería igual a:

$$Req3 = E + V(\log(V) + 1)$$

## 12.5 Cuarto Requerimiento

Para el cuarto requerimiento, con el fin de encontrar red de expansión mínima. Utilizamos el algoritmo Prim de la librería, cuya complejidad es  $E \log(V)$  en el caso una lista de adyacencias con binary heap que corresponde a nuestro modelo de grafo. Posteriormente, para hallar la distancia total del MST se recorrió la estructura DistTo sumando los pesos de los arcos con una complejidad de  $E$ . Asimismo, para la rama más larga del MST ya que la estructura MST no fue creada por errores en la librería, nos vimos en la necesidad de crear un grafo auxiliar. El cual recorría la estructura edgeTo para que luego realizando BFS se pudiera encontrar la rama más larga, lo anterior tiene una complejidad de  $V + E$ . En total, su orden de crecimiento sería igual a:

$$Req4 = V + E(\log(V) + 2)$$

## 12.6 Quinto Requerimiento

Para dar solución a este requerimiento, encontramos los vértices adyacentes al landing point dado, lo cuál se demora  $O(1)$  ya que nuestro grafo utiliza una lista de adyacencia como estructura de datos. Luego, para los vértices encontrados se vuelven a encontrar sus adyacentes en otra vez  $O(1)$  para cada vértice, los vértices encontrados pasarán por el mismo paso con el fin de asegurar de encontrar los adyacentes que necesitamos. Por lo tanto, teniendo en cuenta que un subgrupo de vértices filtrados es igual a  $V'$  en su caso promedio. En total, su orden de crecimiento sería igual a:

$$Req5 = V' + (V')^2 + (V')^3$$

## 12.7 Sexto Requerimiento

En primera instancia, es importante resaltar que el ejemplo no cumple con las instrucciones dadas ya que no existe una conexión directa entre un landing point en Cuba y otro en Venezuela por medio del cable ALBA-1. Por lo tanto, asumimos que el ejercicio buscaba retornar todos los landing points conectados al cable dado por parámetro. Por lo tanto, para este requerimiento hicimos uso del mapa de cable-LP, el cual tiene de llaves a los cables y como valor a todos los landing points conectados a ese cable. Por lo tanto, su orden de crecimiento corresponde a la iteración del número de cables presentes  $C$ . En total, su orden de crecimiento sería igual a:

$$Req6 = C$$

## 12.8 Séptimo Requerimiento

Similar al tercer requerimiento, con el fin de hallar la ruta mínima entre dos IPs, usamos la librería de ipapi para obtener los países correspondientes y posterior a esto utilizar el algoritmo de dijkstra que encuentra los caminos mínimos para un vértice (en nuestro caso el vértice de origen). El orden de crecimiento correspondiente a este es igual a  $E + V \log(V)$ . Ahora bien, para hallar el camino entre el vértice de origen y el de destino usamos la función de pathTo debe iterar los mapas creados por dijkstra que en su peor caso tiene un orden de crecimiento igual a  $V$ . En total, su orden de crecimiento sería igual a:

$$Req7 = E + V(\log(V) + 1)$$

## 12.9 Octavo Requerimiento

Para graficar los mapas, es necesario calcular individualmente su orden de crecimiento en el caso de cada uno de los requerimientos:

- **Requerimiento 1:** Para hallar la ruta entre los dos vértices fuertemente conectados, decidimos graficar la ruta mínima usando el algoritmo de dijkstra que encuentra los caminos mínimos para un vértice (en nuestro caso el vértice de origen). El orden de crecimiento correspondiente a este es igual a  $E + V \log(V)$ . Ahora bien, para hallar el camino entre los dos vértices usamos la función de pathTo debe iterar los mapas creados por dijkstra que en su peor caso tiene un orden de crecimiento igual a

$V$ . Finalmente, iteramos el camino obtenido para situar los marcadores y las líneas, lo que es igual a  $DJ$ . En total, su órden de crecimiento sería igual a:

$$Req8.1 = E + V(\log(V) + 1) + DJ$$

- **Requerimiento 2:** Iteramos los adyacentes del vértice con mayor arco para generar los marcadores y las conexiones entre el vértice y cada uno de sus adyacentes esto se hace en  $V'$ . En total, su órden de crecimiento sería igual a:

$$Req8.2 = V'$$

- **Requerimiento 3:** Iteramos el camino obtenido por pathTo para situar los marcadores y las líneas, lo que es igual a  $DJ$ . En total, su órden de crecimiento sería igual a:

$$Req8.3 = DJ$$

- **Requerimiento 4:** Recorremos los vértices del grafo para situar los marcadores, lo que se demora  $V$  y para las líneas recorremos los arcos, que tiene una duración de  $E$ . En total, su órden de crecimiento sería igual a:

$$Req8.4 = V + E$$

- **Requerimiento 5:** Recorremos la lista de países con vértices afectados, lo que tiene una duración de  $V' + (V')^2 + (V')^3$  ya que es el tamaño de la lista obtenida en el requerimiento 5. En total, su órden de crecimiento sería igual a:

$$Req8.5 = V' + (V')^2 + (V')^3$$

## 13 Resultados

Analizador		Requerimiento 1
Notacion Tilda	$\sim N(CTS + LP + CNN)$	$\sim N(5 + V + E)$
Big Theta	$\Theta(CTS + LP + CNN)$	$\Theta(V + E)$
Big O	$O(CTS + LP + CNN)$	$O(V + E)$
Big Omega	$\Omega(CTS + LP + CNN)$	$\Omega(V + E)$
Requerimiento 2		Requerimiento 3
Notacion Tilda	$\sim N(LP)$	$\sim N(E + V(\log(V) + 1))$
Big Theta	$\Theta(LP)$	$\Theta(E + V\log(V))$
Big O	$O(LP)$	$O(E + V\log(V))$
Big Omega	$\Omega(LP)$	$\Omega(E + V\log(V))$
Requerimiento 4		Requerimiento 5
Notacion Tilda	$\sim N(V + E(\log(V) + 2))$	$\sim N(V' + (V')^2 + (V')^3)$
Big Theta	$\Theta(V + E(\log(V)))$	$\Theta(V' + (V')^2 + (V')^3)$
Big O	$O(V + E(\log(V)))$	$O(V'_w + (V'_w)^2 + (V'_w)^3)$
Big Omega	$\Omega(V + E(\log(V)))$	$\Omega(V'_B + (V'_B)^2 + (V'_B)^3)$
Requerimiento 6		Requerimiento 7
Notacion Tilda	$\sim N(C)$	$\sim N(E + V(\log(V) + 1))$
Big Theta	$\Theta(C)$	$\Theta(E + V\log(V))$
Big O	$O(C_w)$	$O(E + V\log(V))$
Big Omega	$\Omega(C_B)$	$\Omega(E + V\log(V))$
Requerimiento 8.1		Requerimiento 8.2
Notacion Tilda	$\sim N(E + V(\log(V) + 1) + DJ)$	$\sim N(V')$
Big Theta	$\Theta(E + V(\log(V) + 1) + DJ)$	$\Theta(V')$
Big O	$O(E + V(\log(V) + 1) + DJ_w)$	$O(V'_w)$
Big Omega	$\Omega(E + V(\log(V) + 1) + DJ_B)$	$\Omega(V'_B)$
Requerimiento 8.3		Requerimiento 8.4
Notacion Tilda	$\sim N(DJ)$	$\sim N(V + E)$
Big Theta	$\Theta(DJ)$	$\Theta(V + E)$
Big O	$O(DJ_w)$	$O(V + E)$
Big Omega	$\Omega(DJ_B)$	$\Omega(V + E)$
Requerimiento 8.5		
Notacion Tilda	$\sim N(V' + (V')^2 + (V')^3)$	
Big Theta	$\Theta(V' + (V')^2 + (V')^3))$	
Big O	$O(V'_w + (V'_w)^2 + (V'_w)^3))$	
Big Omega	$\Omega(V'_B + (V'_B)^2 + (V'_B)^3)$	