Reto 4 EDA

Alejandra Melo
a.melo4@uniandes.edu.co
código: 202021526

Datos importantes:
-Connections fue considerado como N
-Landing points equivale a 1/3N
-Countries corresponde a 1/14N

Req. 1:

```
def getClustCom(analyzer, lp1, lp2):
    """
    Retorna el número total de clústeres presentes en
    la red e informa si los landing points están en el
    mismo clúster o no.
    """
    sccs = scc.KosarajuSCC(analyzer["connections"])

    #recorrer tabla de landing points y buscar los ids
    id_lp1 = VNombreaNum(analyzer, lp1)
    id_lp2 = VNombreaNum(analyzer, lp2)
    vert = gr.vertices(analyzer["connections"])
    e1 = False
    e2 = False

    for v in lt.iterator(vert):
        if e1 == True and e2 == True:
            break
        num = str.split(v,"-")
        if id_lp1 == num[0]:
            vertice1 = v
            e1 = True
        elif id_lp2 == num[0]:
            vertice2 = v
            e2 = True

    clusters = scc.connectedComponents(sccs)
    mismo_c = scc.stronglyConnected(sccs, vertice1, vertice2)

    return (clusters, mismo_c)
```

Complejidad: 2 (2/3N) + E => O(2*2/3N)
Tiempo promedio: 1614.998
Memoria promedio: 138.546

Req. 2:

```
def getPuntosConex(analyzer):
    """
    Retorna la lista de landing points (nombre, pais,
    identificador) y el total de cables conectados a
    dichos landing points.
    """
    vert = gr.vertices(analyzer["connections"])
    max = 0
    lista_max = lt.newList('ARRAY_LIST')
    for lp in lt.iterator(mp.valueSet(analyzer["landing_points"])):
        l_v = lt.newList('ARRAY_LIST')
        for v in lt.iterator(vert):
            num = str.split(v,"-")
            if str(lp["elements"][0]["landing_point_id"]) == num[0]:
                lt.addLast(l_v, v)
        calc_cables = lt.size(l_v)
        if calc_cables > max:
            max = calc_cables
            lista_max = lt.newList('ARRAY_LIST')
            lt.addLast(lista_max, lp["elements"][0]["landing_point_id"])
        elif calc_cables == max:
            lt.addLast(lista_max, lp["elements"][0]["landing_point_id"])

    return(lista_max, max)
```

Complejidad: 2N + 1 => O(2N)
Tiempo promedio: 9148.273
Memoria promedio: 18.555

Req. 3:

```
def getRutaMenorDist(analyzer, paisA, paisB):
    """
    Retorna la ruta (incluir la distancia de conexión [km]
    entre cada par consecutivo de landing points) y la
    distancia total de la ruta.
    """
    #encontrar la capital de cada país
    cap_A = buscaCapital(analyzer, paisA)
    cap_B = buscaCapital(analyzer, paisB)

    #encontrar landing_point de cada capital y lo devuelve como id
    vertA = VNombreaNum(analyzer, cap_A)
    vertB = VNombreaNum(analyzer, cap_B)
    e1 = False
    e2 = False
    vertixA = str(vertA)
    vertixB = str(vertB)
    #recuperar los vértices (landing point-cable)
    vert = gr.vertices(analyzer["connections"])
    for v in lt.iterator(vert):
        if e1 == True and e2 == True:
            break
        num = str.split(v,"-")
        if vertA == num[0]:
            vertixA = str(v)
            e1 = True
        elif vertB == num[0]:
            vertixB = str(v)
            e2 = True

    dij = djk.Dijkstra(analyzer["connections"], vertixA)
    assert (djk.hasPathTo(dij, vertixB) is True)
    path = djk.pathTo(dij, vertixB)
    distancia = djk.distTo(dij, vertixB)

    return(path, distancia)
```

Complejidad: 1.8N + E logN => O(2N + logN)
Tiempo promedio: 1236.645
Memoria promedio: 29.708

Req. 4:

```
def getInfraest(analyzer):
    """
    Identificar la red de expansión minima en cuanto a distancia que
    pueda darle cobertura a la mayor cantidad de landing points.
    Retorna: # de nodos conectados a la red de expansion minima (camino
    de menor costo que conecta la mayor cantidad de nodos), costo total
    de la red de expansión minima y presentar la rama mas larga (mayor
    numero de arcos entre raiz y la hoja que hace parte de la red de expansion minima).
    """
    g_mst = prim.PrimMST(analyzer["connections"])
    peso = prim.weightMST(analyzer["connections"], g_mst)
    lista_v = lt.newList("ARRAY_LIST")
    grafo_mst = gr.newGraph(datastructure='ADJ_LIST', directed=True,
                            size=10000, comparefunction=compareIds)

    for vert in lt.iterator(mp.valueSet(g_mst["edgeTo"])):
        v_origen = vert["vertexA"]
        v_destino = vert["vertexB"]
        costo = vert["weight"]
        if gr.containsVertex(grafo_mst, v_origen) != True:
            gr.insertVertex(grafo_mst, v_origen)
        if gr.containsVertex(grafo_mst, v_destino) != True:
            gr.insertVertex(grafo_mst, v_destino)
    for vert in lt.iterator(mp.valueSet(g_mst["edgeTo"])):
        v_origen = vert["vertexA"]
        v_destino = vert["vertexB"]
        costo = vert["weight"]
        if gr.containsVertex(grafo_mst, v_origen) != True:
            gr.insertVertex(grafo_mst, v_origen)
        if gr.containsVertex(grafo_mst, v_destino) != True:
            gr.insertVertex(grafo_mst, v_destino)
        gr.addEdge(grafo_mst, v_origen, v_destino, int(costo))
        if lt.isPresent(lista_v, v_origen) != True:
            lt.addLast(lista_v, v_origen)
        elif lt.isPresent(lista_v, v_destino) != True:
            lt.addLast(lista_v, v_destino)

    nodos = lt.size(lista_v)
```

Complejidad: N
Tiempo promedio: 5844.423
Memoria promedio: 24.229

Req. 5:

```
def getFallas(analyzer, lpe):
    """
    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.
    """
    #Se define grafo temporal para carga de datos
    g_pais = gr.newGraph(datastructure='ADJ_LIST',
                                       directed=True,
                                       size=250,
                                       comparefunction=compareIds)

    #Recupera id del LP a partir del LP de entrada
    id_lp = VNombreaNum(analyzer, lpe)
    #Recupera todos los vertices del LP de entrada
    nom_v_lp = numaVertLp(analyzer, id_lp)
    gr.insertVertex(g_pais, lpe)

    for v_a in lt.iterator(nom_v_lp):
        #Recupera los vertices adyacentes de cada vertice del LP
        vert_adj = lt.newList("SINGLE LINKED")
        vert_adj = gr.adjacents(analyzer["connections"], v_a)
        vertixA = v_a
    for v_b in lt.iterator(vert_adj):
        vertixB = v_b
        #Recupera arco de cada vertice adyacente
        arco = gr.getEdge(analyzer["connections"], vertixA, vertixB)
        dist_new = arco["weight"]
        #Recupera pais de vertice adyacente
        id_v_adj = str.split(vertixB,"-")
        for lp in lt.iterator(mp.valueSet(analyzer["landing_points"])):
            if lp["elements"][0]["landing_point_id"] == id_v_adj[0]:
                name = lp["elements"][0]["name"]
                p = str.split(name, ", ")
                tam = len(p)
                pais = p[tam-1]

        if gr.containsVertex(g_pais, pais) != True:
            gr.insertVertex(g_pais, pais)
            gr.addEdge(g_pais,lpe,pais,int(dist_new))
        else:
            #Recupera peso arco existente en grafo
            arco = gr.getEdge(g_pais, lpe, pais)
            dist_ant = arco["weight"]
            if dist_new < dist_ant:
                #Borrar el vértice y volverlo a crear con el nuevo arco
                gr.removeVertex(g_pais, pais)
                gr.insertVertex(g_pais, pais)
```

```
                #Borrar el vértice y volverlo a crear con el nuevo arco
                gr.removeVertex(g_pais, pais)
                gr.insertVertex(g_pais, pais)
                gr.addEdge(g_pais,lpe,pais,int(dist_new))

    return(g_pais)
```

Complejidad: O(3N + 2/3N)
Tiempo promedio: 771.544
Memoria promedio: 251.740

Req. 7:

```
def DatosIP(ip_id):
    api_url = "http://ip-api.com/json/"
    res = rq.get(api_url+ip_id+"?fields=16577")
    api_res = js.loads(res.content)
    return (api_res)

def getMejorRuta(analyzer, ip1, ip2):
    dat_ip1 = DatosIP(ip1)
    pais_ip1 = dat_ip1["country"]
    lon_ip1 = dat_ip1["lon"]
    lat_ip1 = dat_ip1["lat"]

    dat_ip2 = DatosIP(ip2)
    pais_ip2 = dat_ip2["country"]
    lon_ip2 = dat_ip2["lon"]
    lat_ip2 = dat_ip2["lat"]

    #Crear copia grafo connections
    c_grafo = gr.newGraph(datastructure='ADJ_LIST',
                          directed=True,
                          size=14000,
                          comparefunction=compareIds)
    c_grafo = analyzer["connections"]
```

```
    #Recupera Capital de ambos paises
    cap_ip1 = buscaCapital(analyzer,pais_ip1)
    cap_ip2 = buscaCapital(analyzer,pais_ip2)

    #Recupera id landing point de cada capital
    ver_cap1 = str(VNombreaNum(analyzer, cap_ip1))
    ver_cap2 = str(VNombreaNum(analyzer, cap_ip2))

    #Recupera ubicación geografica de capitales

    e1 = 0
    e2 = 0
    for country in lt.iterator(mp.valueSet(analyzer["countrys"])):
        if pais_ip1 == country["elements"][0]["CountryName"] and e1 == 0:
            lat_cap1 = country["elements"][0]["CapitalLatitude"]
            lon_cap1 = country["elements"][0]["CapitalLongitude"]
            e1 = 1
        elif pais_ip2 == country["elements"][0]["CountryName"] and e2 == 0:
            lat_cap2 = country["elements"][0]["CapitalLatitude"]
            lon_cap2 = country["elements"][0]["CapitalLongitude"]
            e2 = 1

    #Calcular distancia ips con capital de pais al que pertenece cada uno
    dist_ip1 = Haversine(lat_ip1,lon_ip1,lat_cap1,lon_cap1)
    dist_ip2 = Haversine(lat_ip2,lon_ip2,lat_cap2,lon_cap2)
```

```
    #agregar vertices ip1 e ip2 a grafo
    gr.insertVertex(c_grafo,ip1)
    gr.insertVertex(c_grafo,ip2)

    #conectar con cada capital
    gr.addEdge(c_grafo, ip1, ver_cap1, weight = dist_ip1)
    gr.addEdge(c_grafo, ip2, ver_cap2, weight = dist_ip2)

    #calcular el camino mas corto
    dij = djk.Dijkstra(c_grafo, ip1)
    assert (djk.hasPathTo(dij, ip2) is True)
    path = djk.pathTo(dij, ip2)
    saltos = st.size(path)

    return(path,saltos)

def DistGeoLandP(analyzer, origen, destino):
    #1)Obtener la latitud y longitud del origen y destino
    for lps in lt.iterator(mp.valueSet(analyzer["landing_points"])):
        if str(origen) in str(lps["elements"][0]["landing_point_id"]):
            lat_o = lps["elements"][0]["latitude"]
            long_o = lps["elements"][0]["longitude"]
        if str(destino) in str(lps["elements"][0]["landing_point_id"]):
            lat_d = lps["elements"][0]["latitude"]
            long_d = lps["elements"][0]["longitude"]
```

```
        if str(destino) in str(lps["elements"][0]["landing_point_id"]):
            lat_d = lps["elements"][0]["latitude"]
            long_d = lps["elements"][0]["longitude"]
    distancia = Haversine(lat_o,long_o,lat_d,long_d)
    return(distancia)

def Haversine(lat_o, long_o, lat_d, long_d):
    #2)Aplicar la función Haversine para calcular las distancias
    rad = mt.pi/180
    f_lat_o = float(lat_o)
    f_lat_d = float(lat_d)
    f_long_o = float(long_o)
    f_long_d = float(long_d)
    dif_lat = f_lat_o - f_lat_d
    dif_long = f_long_o - f_long_d
    r_tierra = 6372.795477598
    a = (mt.sin(rad*dif_lat)/2)**2 + mt.cos(rad*f_lat_d) * mt.cos(rad*f_lat_o) * (mt.sin(
    distancia = 2 * r_tierra * mt.asin(mt.sqrt(a))
    return(distancia)
```

Complejidad: 0.87N + E logN => O(N + logN)
Tiempo promedio:
Memoria promedio: