Documento de Análisis del Reto 4

Nombre: Julián Galindo Mora

Código: 202111477

Correo: j.galindom2@uniandes.edu.co

Diciembre 2021

1. Descripción general:

En primera instancia es importante destacar que para la elaboración de este reto se optó por desarrollar un algoritmo en el que se diera prioridad a los tiempos de respuestas de las consultas y requerimientos por encima de todo. Para ello se elaboró un algoritmo de carga que, si bien cuenta con un tiempo de ejecución medianamente elevado, permite crear estructuras de datos ajenas a las recomendadas por las instrucciones de este reto, con el fin de manipular los datos de una manera óptima y precisa, facilitando la obtención de la información requerida. Estas estructuras están constituidas principalmente por una serie de mapas que organizan y catalogan información precisa para cada uno de los requerimientos previstos, donde se podría encontrar entre ellos la relación "CódigoIata - Aeropuerto", "Trayecto - Distancia", entre otros.

En cuanto a la medición de tiempos desarrollada en este reto, se decidió por determinar el tiempo de ejecución de una función como la diferencia de tiempo que existe desde la ejecución de una función de requerimiento en el archivo vista, hasta la elaboración o desarrollo de su última línea (siendo esta normalmente aquella que daba las instrucciones de impresión de tablas a través del módulo "tabulate", el cual también fue requerido para la correcta elaboración y visualización de la información deseada por cada parámetro). Además, el tiempo de ejecución fue determinado por la importación del módulo "timeit" en Python que fue implementado únicamente en el archivo de vista por la aclaración previamente descrita.

Finalmente, es importante destacar que para los archivos de 80% y large, el límite de recursión que utilizaban los grafos excede el que está establecido como predeterminado en Python, por ello se aumentó este límite con la ayuda de la función "setrecursionlimit" del módulo "sys" hasta el valor de 1048576 para su correcto funcionamiento.

2. **Análisis de complejidad por requerimiento:**

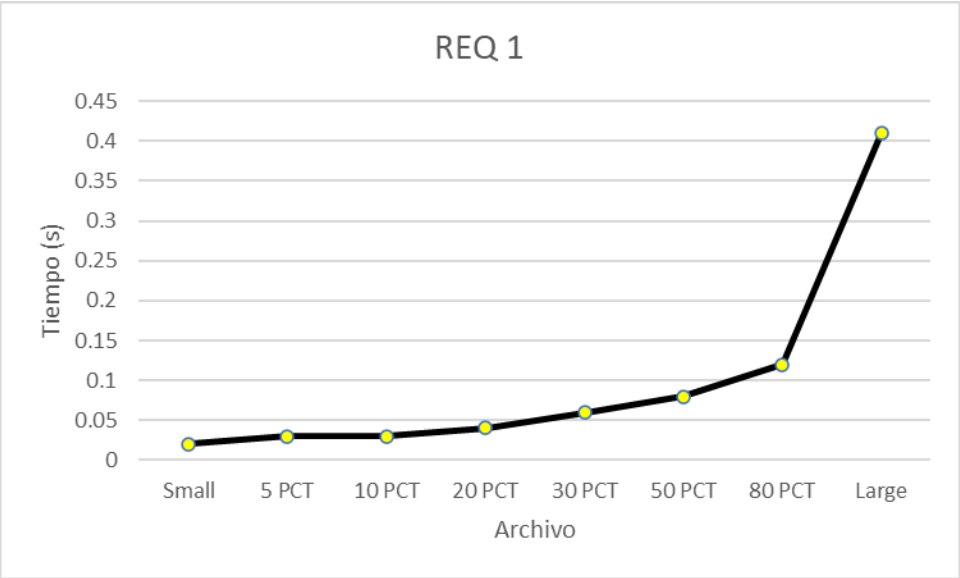
2.1. Requerimiento 1:

- **TADs usados:** TAD Lista (single linked), TAD Mapa (linear probing), TAD Grafo (adj list)
- **Relación con la carga:** Este requerimiento solo usó dos de las estructuras implementadas por la carga: el grafo “vuelos” y el mapa “iataInfo”, los cuales contenían la relación entre códigos IATA y rutas, y la información de cada código IATA respectivamente.
- **Descripción:** Este requerimiento buscaba encontrar los 5 aeropuertos que tuvieran un mayor flujo o cantidad de conexiones dentro de todos los vuelos posibles. Teniendo a esta cantidad como la suma del número de aeropuertos a los que era posible llegar desde el aeropuerto especifico, con el número de aeropuertos desde los que se podía llegar hasta el aeropuerto específico.

- **Código:**

```
def interconection (analyzer):
    iataList = m.keySet(analyzer["iataInfo"])
    table = []
    for iata in lt.iterator(iataList):
        inNumb = int(gr.indegree(analyzer["vuelos"], iata))
        outNumb = int(gr.outdegree(analyzer["vuelos"], iata))
        numVertex = inNumb + outNumb
        line = []
        path = m.get(analyzer["iataInfo"], iata)
        values = me.getValue(path)
        nombre = lt.getElement(values, 1)
        ciudad = lt.getElement(values, 2)
        pais = lt.getElement(values, 3)
        line.append(iata)
        line.append(nombre)
        line.append(ciudad)
        line.append(pais)
        line.append(numVertex)
        line.append(inNumb)
        line.append(outNumb)
        if (numVertex > 0):
            table.append(line)
    return table
```

- **Complejidad:** O(n), donde n es el número de aeropuertos.
- **Gráfica:**



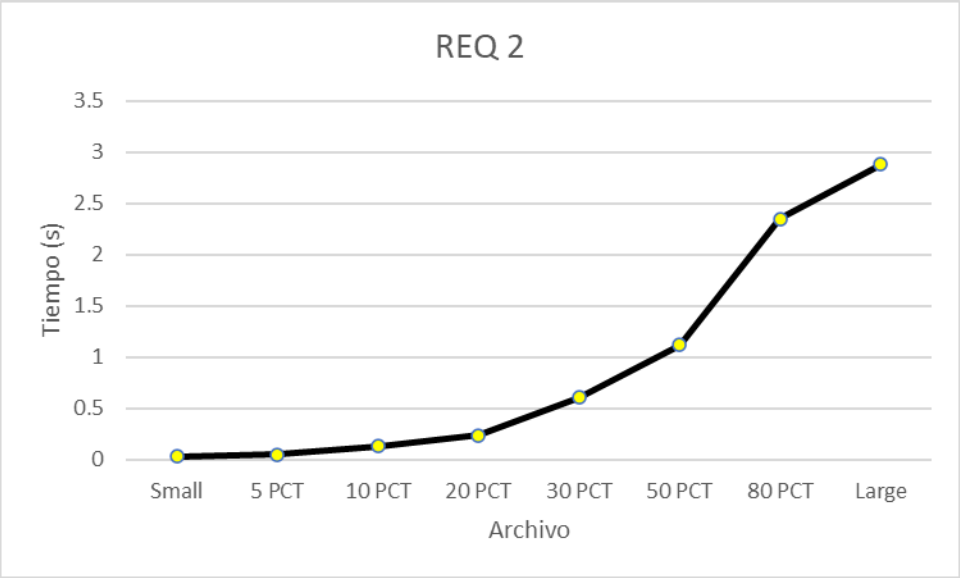
2.2. Requerimiento 2: LED RTP

- **TADs usados:** TAD Lista (single linked), TAD Mapa (linear probing), TAD Grafo (adj list)
- **Relación con la carga:** Este requerimiento solo usó dos de las estructuras implementadas por la carga: el grafo “vuelos” y el mapa “iataInfo”, los cuales contenían la relación entre códigos IATA y rutas, y la información de cada código IATA respectivamente.
- **Descripción:** Este requerimiento buscaba identificar la cantidad de componentes fuertemente conectados que estaban presentes en el grafo “vuelos”. Así mismo, este buscaba identificar si un par de aeropuertos (vértices) estaban ubicados dentro de un mismo componente conectado.
- **Caso de Prueba:** Para probar este requerimiento se usaron como casos de prueba los códigos IATA: “LED” y “RTP”, que correspondían a los aeropuertos de “Pulkovo, Rusia” y “Rutland, Australia” respectivamente.

- **Código:**

```
def clusteres (analyzer, iataAp1, iataAp2):
    kosaraju = scc.KosarajuSCC(analyzer["vuelos"])
    components = scc.connectedComponents(kosaraju)
    connection = scc.stronglyConnected(kosaraju, iataAp1, iataAp2)
    answer = (components, connection)
    return answer
```

- **Complejidad:** O(n)
- **Gráfica:**



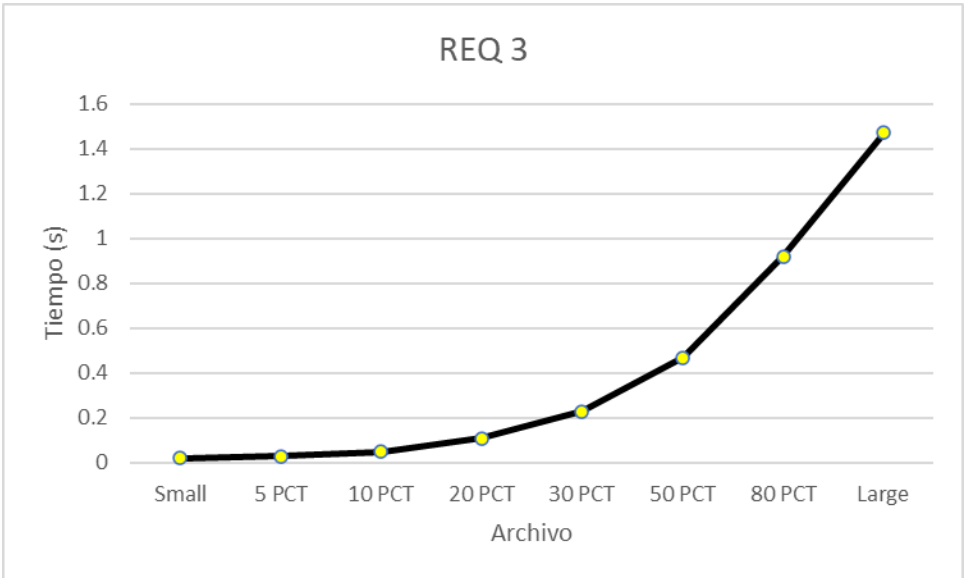
2.3. Requerimiento 3:

- **TADs usados:** TAD Lista (single linked), TAD Mapa (linear probing), TAD Grafo (adj list)
- **Relación con la carga:** Este requerimiento solo usó dos de las estructuras implementadas por la carga: el grafo “vuelos” y el mapa “iataInfo”, los cuales contenían la relación entre códigos IATA y rutas, y la información de cada código IATA respectivamente.
- **Descripción:** Este requerimiento consistía en hallar la ruta de vuelos que se debía seguir desde la ciudad de origen hasta la ciudad de destino, teniendo en cuenta que la ruta debía ser la más corta posible en términos de distancia. De esta manera se debía indicar por cuales aeropuertos o paradas había que parar y cuanto era la distancia total recorrida.

- **Caso de Prueba:** Para el desarrollo de las pruebas de este parámetro se usaron como entradas a las ciudades de “Saint Petersburg” y “Lisbon” (Portugal) como origen y destino respectivamente.
- **Código:**

```
def shortestRoute (analyzer, origin, destiny):
    dijkstra = djk.Dijkstra(analyzer["vuelos"], origin)
    recorrido = djk.pathTo(dijkstra, destiny)
    flightList = []
    for i in lt.iterator(recorrido):
        flightList.append(i)
    j = 0
    table1 = []
    route = []
    for dic in flightList:
        linea = []
        linea.append(dic["vertexA"])
        linea.append(dic["vertexB"])
        linea.append(dic["weight"])
        table1.append(linea)
        if j == 0:
            route.append(dic["vertexA"])
            route.append(dic["vertexB"])
        else:
            route.append(dic["vertexB"])
        j += 1
    table2 = []
    for iata in route:
        pair = m.get(analyzer["iataInfo"], iata)
        value = me.getValue(pair)
        k = 0
        linea2 = [iata]
        for element in lt.iterator(value):
            if k < 3:
                linea2.append(element)
            k += 1
        table2.append(linea2)
    answerTuple = (table1, table2)
    return answerTuple
```

- **Complejidad:** O(v*e), donde v es el número de aeropuertos y e el número de vuelos.
- **Gráfica:**



2.4. Requerimiento 4:

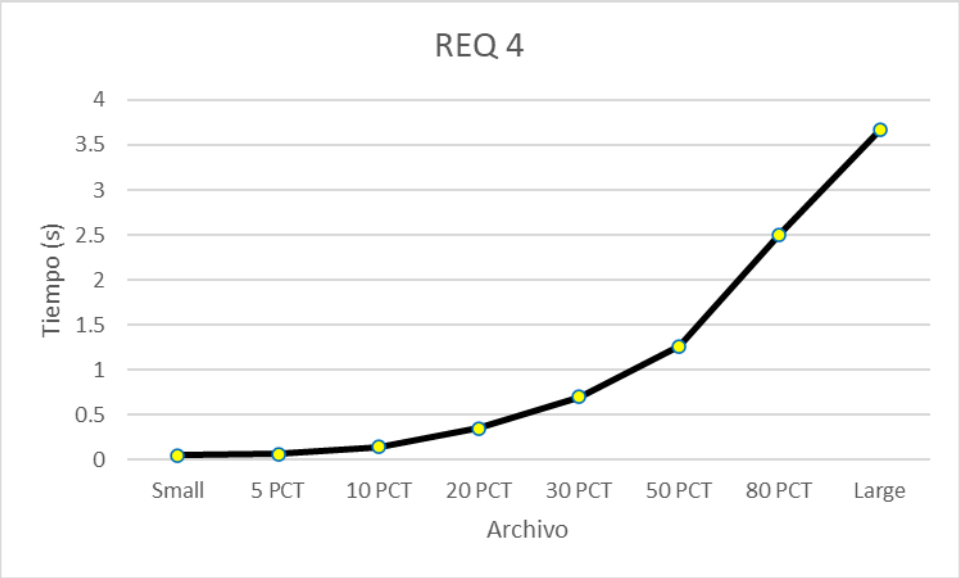
- **TADs usados:** TAD Lista (single linked), TAD Mapa (linear probing), TAD Grafo (adj list)
- **Relación con la carga:**
- **Descripción:** Este requerimiento busca hallar el MST que puede existir dentro del grafo, para de esta manera identificar una ruta óptima para ir a todos los vértices sin importar un origen en específico.
- **Código:**

```
def travelerMiles (analyzer, miles):
    prim = pr.PrimMST(analyzer["doubleRoutes"])
    sub1 = prim["edgeTo"]
    sub2 = sub1["table"]
    sub3 = sub2["elements"]
    iataList = []
    table2 = []
    for dic in sub3:
        llave = me.getKey(dic)
        line2 = []
        if llave != None:
            valor = me.getValue(dic)
            line2.append(valor["vertexA"])
            if valor["vertexA"] not in iataList:
                iataList.append(valor["vertexA"])
            line2.append(valor["vertexB"])
            if valor["vertexB"] not in iataList:
                iataList.append(valor["vertexB"])
            line2.append(valor["weight"])
            table2.append(line2)
    table1 = []
    for iata in iataList:
        pair = m.get(analyzer["iataInfo"], iata)
        value = me.getValue(pair)
        i = 0
        line1 = [iata]
        for element in lt.iterator(value):
            if i < 3:
                line1.append(element)
            i += 1
        table1.append(line1)

    answer = (table1, table2)
    return answer
```

- **Complejidad:** O(n*m), donde n es el número de aeropuertos y m el número de vuelos.

- **Gráfica:**



2.5. Requerimiento 5:

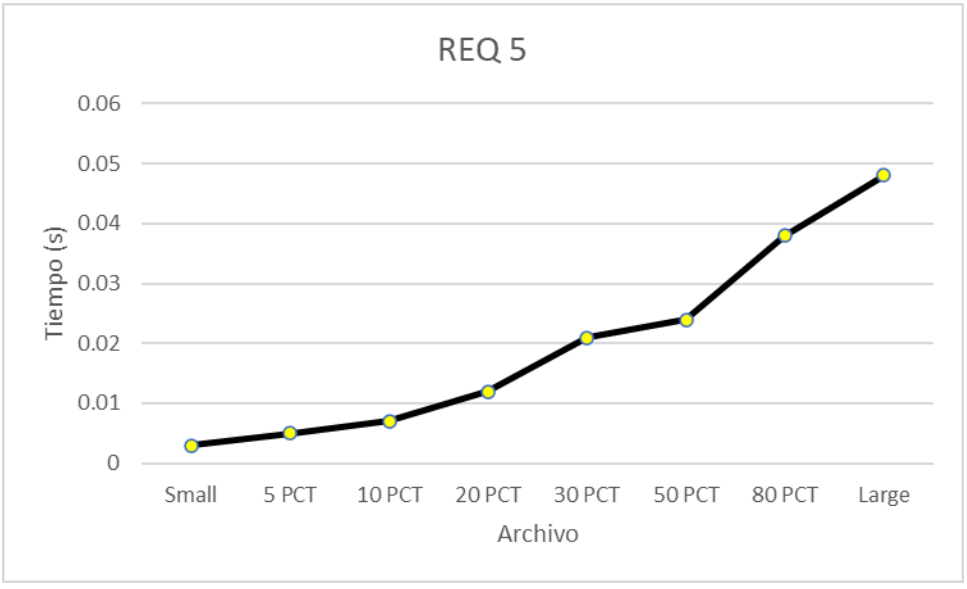
- **TADs usados:** TAD Lista (single linked), TAD Mapa (linear probing), TAD Grafo (adj list)
- **Relación con la carga:** Este requerimiento usó tres de las estructuras implementadas por la carga: el grafo “vuelos”, el grafo “doubleRoutes” y el mapa “iataInfo”, los cuales contenían la relación entre códigos IATA y rutas (para cada tipo de grafo), y la información de cada código IATA respectivamente.
- **Descripción:** Este requerimiento buscaba encontrar el impacto que tendría en los vuelos a nivel global (grafo “vuelos”), si el aeropuerto ingresado como parámetro fuera cerrado o pues en mantenimiento por un lapso temporal. Donde se destacaría el cambio en el número de aeropuertos (vértices) y vuelos (arcos) disponibles.
- **Caso de Prueba:** Para la prueba de este requerimiento se evaluó al aeropuerto Internacional de Dubái, identificado con el código IATA DXB, como aquél que sería cerrado.

- **Código:**

```
def closedEffect (analyzer, closedIata):
    table = []
    orDiVe = gr.numVertices(analyzer["vuelos"])
    orDiEd = gr.numEdges(analyzer["vuelos"])
    orGrVe = gr.numVertices(analyzer["doubleRoutes"])
    orGrEd = gr.numEdges(analyzer["doubleRoutes"])
    n1 = gr.adjacents(analyzer["vuelos"], closedIata)
    apDiVe = lt.size(n1)
    apDiEd = gr.degree(analyzer["vuelos"], closedIata)
    n2 = gr.adjacents(analyzer["doubleRoutes"], closedIata)
    apGrVe = lt.size(n2)
    apGrEd = gr.degree(analyzer["doubleRoutes"], closedIata)
    iataAffected = []
    for j in lt.iterator(n2):
        if j not in iataAffected:
            iataAffected.append(j)
    numAffected = len(iataAffected)
    for iata in iataAffected:
        pair = m.get(analyzer["iataInfo"], iata)
        value = me.getValue(pair)
        linea = [iata]
        i = 0
        for element in lt.iterator(value):
            if i > 2:
                break
            linea.append(element)
            i+=1
        table.append(linea)
    originals = (orDiVe, orDiEd, orGrVe, orGrEd)
    airports = (apDiVe, apDiEd, apGrVe, apGrEd)
    answer = (table, originals, airports, numAffected)

    return answer
```

- **Complejidad:** O(v*e), donde v es el número de vértices y e el número de arcos.
- **Gráfica:**



3. **Funciones Auxiliares:**

3.1. Escoger Ciudad:

- **Descripción:** Debido a que existen muchas ciudades llamadas igual, esta función permite escoger a través de una tabla cual de todas las ciudades que existen con el nombre del parámetro ingresado deseas seleccionar para los requerimientos que necesiten de una ciudad especifica como parámetro. Cabe destacar que la tabla le proporciona al usuario toda la información correspondiente a cada ciudad con el fin de que este pueda distinguir con claridad a cuál hace referencia.
- **Relación con la carga:** Para el funcionamiento de esta función se usó el mapa "cityInfo" que tenía como llaves una unión del nombre de la ciudad con su respectivo id y como valor: una lista de que contenía la información de cada ciudad como su latitud, longitud, país, etc.
- **Código:**

```
def chooseCity (analyzer, city):
    llaves = m.keySet(analyzer["cityInfo"])
    table = []
    for key in lt.iterator(llaves):
        line = []
        ind = len(table) + 1
        chainList = key.split("-")
        nombre = chainList[0]
        if nombre == city:
            path = m.get(analyzer["cityInfo"], key)
            valList = me.getValue(path)
            line.append(ind)
            line.append(nombre)
            for val in lt.iterator(valList):
                line.append(val)
            table.append(line)

    return table
```

- **Complejidad:** O(n), donde n es el número de ciudades.

3.2. Aeropuerto más Cercano:

- **Descripción:** Esta función buscaba encontrar el aeropuerto existente más cercano a una ciudad especifica que el usuario ingresaba como parámetro con el fin de poder determinar posibles rutas entre ciudades.
- **Código:** O(n Log n) donde n es el **número de ciudades.**

```
def shortestAirport (analyzer, city):
    pair = m.get(analyzer["cityInfo"], city)
    value = me.getValue(pair)
    i = 0
    city_lat = None
    city_lon = None
    for element in lt.iterator(value):
        if i == 2:
            city_lat = element
        if i == 3:
            city_lon = element
        i += 1
    cityLoc = (float(city_lat), float(city_lon))
    org_matrix = sorted(analyzer["iataLocation"], key=lambda x:x[1])
    distanceDict = {}
    for airport in org_matrix:
        airLat = airport[1]
        airLon = airport[2]
        airTup = (float(airLat), float(airLon))
        distance = haversine(cityLoc, airTup)
        distanceDict[airport[0]] = distance
    minIata = None
    minDis = 999999999
    for iata in distanceDict:
        disValue = distanceDict[iata]
        if disValue < minDis:
            minDis = disValue
            minIata = iata
    wishedPair = m.get(analyzer["iataInfo"], minIata)
    wishedValue = me.getValue(wishedPair)
    wishedAirport = [minIata]
    for j in lt.iterator(wishedValue):
        wishedAirport.append(j)
    return wishedAirport
```

- **Complejidad:** O(n*m^2), donde n es el número de ciudades y m el número aeropuertos.

4. Información Adicional:

4.1. Ambiente de Pruebas:

	Maquina 1
Procesador	AMD Ryzen 7 4800H with Radeon Graphics 2.90 GHz
Memoria RAM (GB)	8 GB
Sistema Operativo	Windows 11 Home Single Language

4.2. Carga de Datos:

- Para la carga de archivos y la construcción del grafo dirigido “vuelos” en el cual se encontraban todos los códigos IATA de todos los aeropuertos provistos por el archivo “airports”; se optó por dejar a los vértices únicamente como el código IATA correspondiente a cada aeropuerto pues este indicador es único. Además, gracias a esta decisión fue mucho más fácil poder implementar todos los arcos (vuelos) propuestos por el archivo “routes”, donde solo se debían identificar el IATA de salida y llegada para conectar dichos vértices. En cuanto a su complejidad, se podría catalogar a este algoritmo como un O(n) siendo n la suma de las cantidades de aeropuertos y rutas disponibles en el archivo seleccionado.

```python
def add_info (analyzer, airport):
    intlist0 = lt.newList()
    lt.addLast(intlist0, airport["Name"].upper())
    lt.addLast(intlist0, airport["City"].upper())
    lt.addLast(intlist0, airport["Country"].upper())
    lt.addLast(intlist0, float(airport["Longitude"]))
    lt.addLast(intlist0, float(airport["Latitude"]))
    m.put(analyzer["iataInfo"], airport["IATA"], intlist0)
    intlist1 = lt.newList()
    m.put(analyzer["routeMap"], airport["IATA"], intlist1)
    gr.insertVertex(analyzer["vuelos"], airport["IATA"])
    line = [airport["IATA"], airport["Latitude"], airport["Longitude"]]
    analyzer["iataLocation"].append(line)

def add_edge (analyzer, route):
    b1= route["Departure"]
    b2= route["Destination"]
    b3= float(route["distance_km"])
    if b1 != None and b2 != None and b3 != None:
        gr.addEdge(analyzer["vuelos"], b1, b2, b3)

    pair = m.get(analyzer["routeMap"], route["Departure"])
    value = me.getValue(pair)

    if lt.isPresent(value, route["Destination"]) == 0:
        lt.addLast(value, route["Destination"])

    joinKey = route["Departure"] + "-" + route["Destination"]
    m.put(analyzer["distances"], joinKey, route["distance_km"])

def add_city (analyzer, city):
    joinKey = city["city_ascii"].upper() + "-" + city["id"]
    infoList = lt.newList()
    lt.addLast(infoList, city["country"].upper())
    lt.addLast(infoList, city["population"].upper())
    lt.addLast(infoList, city["lat"])
    lt.addLast(infoList, city["lng"])
    lt.addLast(infoList, city["id"])
    m.put(analyzer["cityInfo"], joinKey, infoList)
```

Funciones add_info, add_edge y add_city: usadas para crear el grafo dirigido y demás estructuras presentes en este reto.

- Por otro lado, para el desarrollo del segundo grafo (no dirigido) “doubleRoutes”, es preciso aclarar que se construyó este grafo con la premisa de que solo se incluirían en él los aeropuertos que tuvieran una ruta de ida y vuelta con cualquier otro aeropuerto (vértices) y que solo se incluirían aquellos vuelos que se pudieran identificar como de ida y vuelta entre dos aeropuertos (arcos). En cuanto a su desarrollo, este se fundamentó a partir de la ejecución de la función double_check. Esta basaba su búsqueda en tres mapas generados por la propia carga de archivos: “iataInfo”, “routeMap” y “distances”. El primero únicamente servía para recorrer sobre llaves que en este caso contenían todos los códigos IATA provistas por el archivo “airports”; mientras que el segundo contenía información más compleja y relevante. Este último tenía como llave el código IATA de cierto aeropuerto en específico, y como valor poseía una lista que contenía los códigos IATA de todos los aeropuertos a donde se podía llegar desde este aeropuerto

original. De esta manera es más fácil verificar si un par de aeropuertos tenían vuelos entre ellos, pero con direcciones contrarias. Por último, el mapa "distances" tenía como llave la unión del IATA de origen y el de destino con un carácter "-" entre ellos, y como valor la distancia que requería ese trayecto: siendo estos valores muy importantes pues fueron gracias a estos valores que se pudo identificar de manera mucho más rápida el peso de los arcos que se necesitaban añadir al grafo.

- Si bien en primera instancia se podría pensar que este algoritmo tiene una complejidad de $O(n^2)$ pues requiere visitar todos los elementos del mapa "routeMap" y a su vez todos los valores de esa lista interna (donde se repetiría una instrucción por los aeropuertos ya añadidos): gracias a las últimas cuatro líneas es posible disminuir esta complejidad hasta $O(n^{3/2})$ pues si bien tiene que seguir recorriendo todos los elementos del mapa, ya no tendrá que comparar, verificar o analizar los vértices ya añadidos al grafo en cada iteración. No obstante, también es importante aclarar que, por alguna razón desconocida, cuando se ejecuta la carga algunas veces es necesario quitar o desactivar las últimas dos líneas pues generar un error desconocido. Haciendo que estos archivos la complejidad sea de $O(n^2)$ y que su tiempo de ejecución se distinto.

```
def double_check(analyzer):
    iataList = m.keySet(analyzer["iataInfo"])
    for dep in lt.iterator(iataList):
        pair1 = m.get(analyzer["routeMap"], dep)
        value1 = me.getValue(pair1)
        for des in lt.iterator(value1):
            pair2 = m.get(analyzer["routeMap"], des)
            value2 = me.getValue(pair2)
            if lt.isPresent(value2, dep) != 0:
                if gr.containsVertex(analyzer["doubleRoutes"], dep) == False:
                    gr.insertVertex(analyzer["doubleRoutes"], dep)
                    if lt.isPresent(analyzer["doubleList"], dep) == 0:
                        lt.addLast(analyzer["doubleList"], dep)
                if gr.containsVertex(analyzer["doubleRoutes"], des) == False:
                    gr.insertVertex(analyzer["doubleRoutes"], des)
                    if lt.isPresent(analyzer["doubleList"], des) == 0:
                        lt.addLast(analyzer["doubleList"], des)
                join_key = dep + "-" + des
                distance_pair = m.get(analyzer["distances"], join_key)
                distance_val = me.getValue(distance_pair)
                gr.addEdge(analyzer["doubleRoutes"], dep, des, distance_val)
                pos_del1 = lt.isPresent(value1, des)
                pos_del2 = lt.isPresent(value2, dep)
                lt.deleteElement(value1, pos_del1)
                lt.deleteElement(value2, pos_del2)
```

Función double_check, usada para crear el grafo no dirigido.

Gráfica de archivos (%) vs tiempo de carga (s)