Documento de Analisis

Req 1

```
def BuildMostConnectedTable(catalog, connectionsmap, top5):
    table=PrettyTable()
    table.field_names = ['Name', 'City', 'Country' , 'IATA', 'connections' , 'Inbound', 'Outbound']
    table.align='l'
    table._max_width= {'Name':15, 'City':10 , 'Country': 15, 'IATA':5, 'Connections':5, 'Inbound':5, 'Outbound':5}
    counter = 0
    for item in lt.iterator(top5):
        entry2= om.get(connectionsmap, item)
        value2 = me.getValue(entry2)
        for element in lt.iterator(value2):
            counter += 1
            codigoIATA = str(element['Name'])
            entry = map.get(catalog['airports'], codigoIATA)
            value = me.getValue(entry)
            table.add_row([value['Name'], str(value['City']), str(value['Country']), codigoIATA,str(element['TotalDegree']), str(
            if counter == 5:
                break
        if counter == 5:
            break
    return table
```

A primera vista podríamos pensar que la función BuildMostConnectedTable aporta la mayor complejidad al requisito al tener un iterador anidado que en condiciones normales haría que la función tuviese complejidad O(n^2). No obstante, podemos ver que dentro de estos iteradores se encuentran dos if que frenan la función cuando counter ==5, es decir, la complejidad de la función termina siendo constante.

```
def MostConnected(graph):
    connectionsmap = om.newMap(omaptype='RBT', comparefunction=CompareTotalDegrees)
    vertices = grph.vertices(graph)
    amountconnected = 0
    for vertex in lt.iterator(vertices):
        dict = {'Name':None, 'indegree': None, 'outdegree': None, 'TotalDegree':None}
        dict['Name'] = str(vertex)
        dict['indegree'] = str(grph.indegree(graph,vertex))
        dict['outdegree'] = str(grph.outdegree(graph,vertex))
        dict['TotalDegree'] = str(int(dict['indegree']) + int(dict['outdegree']))
        if not om.contains(connectionsmap, dict['TotalDegree']):
            newlist = lt.newList(datastructure='ARRAY_LIST')
            lt.addLast(newlist, dict)
            om.put(connectionsmap, str(dict['TotalDegree']), newlist)
        else:
            entry = om.get(connectionsmap, dict['TotalDegree'])
            degreeslist = me.getValue(entry)
            lt.addLast(degreeslist, dict)
        if dict['TotalDegree'] != '0':
            amountconnected += 1
    keys = om.keySet(connectionsmap)
    return amountconnected, keys, connectionsmap
```

La función MostConnected es la que más complejidad aporta al requerimiento ya que contiene un for el cual itera sobre todos los vértices existentes del grafo. De esta forma la complejidad de esta función y la complejidad mayor de este requerimiento es O(v) donde v es el número de vértices del grafo.

Req 2

```
def getcomponents(graph):
    sc = scc.KosarajuSCC(graph)
    return scc.connectedComponents(sc)

def RSC(graph, verta, vertb):
    graphone = scc.KosarajuSCC(graph)
    return scc.stronglyConnected(graphone, verta, vertb)
```

El requerimiento usa principalmente el algoritmo Kosaraju, el cual tiene una complejidad O(3(V+E))

Req 3

```
def Closest_Path(catalog, ciudadorigen, ciudaddestino):
    lat = float(ciudadorigen['lat'])
    lng = float(ciudadorigen['lng'])
    upperlat = float(lat)
    lowerlat = float(lat)
    upperlng = float(lng)
    lowerlng = float(lng)
    airportstree = om.newMap(omaptype='RBT', comparefunction=CompareDistance)
    destinytree = Closest_To_Destiny(catalog, ciudaddestino)
    closestdestiny = om.minKey(destinytree)
    entry = om.get(destinytree, str(closestdestiny))
    destinyIATA = me.getValue(entry)
    condition = True
    while condition:
        upperlat += 0.1
        lowerlat -= 0.1
        upperlng += 0.1
        lowerlng -= 0.1
        lat_in_range = om.keys(catalog['CoordinatesTree'],str(lowerlat), str(upperlat))
        if lat_in_range != None:
            for latitude in lt.iterator(lat_in_range):
                entry = om.get(catalog['CoordinatesTree'], str(latitude))
                longitudeindex = me.getValue(entry)
                for airport in lt.iterator(longitudeindex):
                    IATAcode = airport['IATA']
                    if lowerlng < float(airport['longitude']) < upperlng:
                            city_coordinates = (lat,lng)
                            airport_coordinates = (float(latitude), float(airport['longitude']))
```

```
                    if lowerlng < float(airport['longitude']) < upperlng:
                            city_coordinates = (lat,lng)
                            airport_coordinates = (float(latitude), float(airport['longitude']))
                            distance = haversine(city_coordinates, airport_coordinates)
                            if not om.contains(airportstree, str(distance)):
                                om.put(airportstree, str(distance), str(IATAcode))
        keys = om.keySet(airportstree)
        for item in lt.iterator(keys):
            entry = om.get(airportstree, item)
            codigoIATA = me.getValue(entry)
            search = djk.Dijkstra(catalog['Fullroutes'], codigoIATA)
            if djk.hasPathTo(search, destinyIATA):
                origindict = OrderedDict()
                destinydict = OrderedDict()
                origindict['distance'] = item
                origindict['IATA'] = codigoIATA
                destinydict['distance'] = closestdestiny
                destinydict['IATA'] = destinyIATA
                minpath = djk.pathTo(search, destinyIATA)
                condition = False
    return origindict, destinydict, minpath
```

Para tener un entendimiento completo de la función haremos un recorrido por sus partes. Primero la función crea un cuadrado de búsqueda para encontrar aeropuertos en el área. Se usa un while para que el cuadrado cada vez cubre más espacio hasta encontrar un aeropuerto que tenga conexión con el aeropuerto de destino. La función sabe que se encontró un aeropuerto cuando la lista de llaves de 'lat_in_range' (un rbt cuyas llaves son la latitud de los aeropuertos) en el intervalo impuesto por el cuadrado de búsqueda no está vacía. Al cumplirse esta condición se entra a un iterador anidado que recorre a los aeropuertos en el intervalo de latitud para verificar que estén en el intervalo de longitud, es por esto que se usa un iterador anidado. Como las longitudes y latitudes de los aeropuertos son tan precisas podría argumentarse que cada longitud y latitud es única. No obstante, revisando el archivo a detalle se encontraron varios aeropuertos que comparten latitud o longitud exactas por lo que se tomó la precaución de revisar por más de un aeropuerto en la misma latitud o misma longitud. Sin embargo, estas ocurrencias no son comunes por lo que la mayoría de veces el iterador anidado actuará como un solo iterador. Seguidamente, los aeropuertos encontrados se ponen en un rbt. Luego iteramos sobre las llaves de este rbt (ordenadas de menor a mayor distancia) para verificar si el aeropuerto correspondiente a esa llave tiene ruta al aeropuerto de destino. En caso de que haya ruta se hace dijkstra para obtener el camino de menor peso al vértice destino y se termina el while. Con todo este contexto podemos concluir que la complejidad de la función depende en la cantidad de veces que se tenga que expandir el cuadrado de búsqueda para encontrar un aeropuerto (a esta variable la llamaremos h), de los aeropuertos cuya latitud entre en el intervalo definido por el cuadrado de búsqueda (a esa variable la llamaremos k) y la cantidad de aeropuertos que tienen misma latitud (a esta variable la llamaremos n) De esta forma, la complejidad de la función es O(h*k*n + k) pero se puede aproximar a O(h*k + k) teniendo en cuenta la poca ocurrencia de aeropuertos con misma latitud.

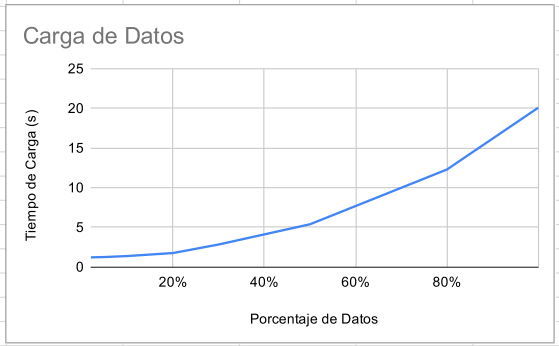
Requerimiento 5:
La función del requerimiento 5 con mayor complejidad temporal es:

```
def listaafectados(catalog,cerrado):
    arcos = grph.edges(catalog['Fullroutes'])
    newlist = lt.newList(datastructure='ARRAY_LIST')
    for item in lt.iterator(arcos): #ITERA Sobre todos los arcos
        vertexA = str(item['vertexA'])
        vertexB = str(item['vertexB'])
        if vertexA == cerrado:
            if not lt.isPresent(newlist, vertexB): #lt.ispresent
complejidad O(N) N=V
                lt.addLast(newlist, vertexB)
        elif vertexB == cerrado:
            if not lt.isPresent(newlist, vertexA):
                lt.addLast(newlist, vertexA)
    return newlist
```

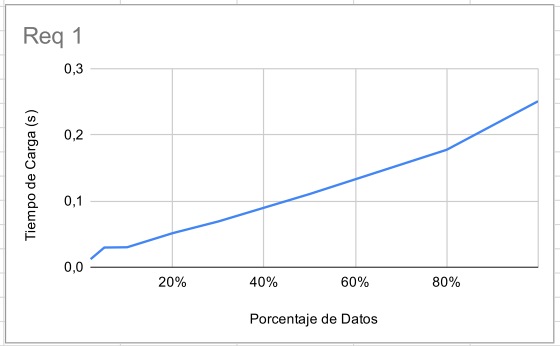
La complejidad temporal de esta función es O(E*V) pues itera sobre todos los arcos que se encuentren en el índice catalog['Fullroutes'] (índice con todos los aeropuertos y rutas aéreas) y además en el peor caso ejecuta siempre la función lt.ispresent() que tiene una

complejidad O(N). Como el máximo número de elementos posible en la lista es V, la complejidad es O(V). Esto se repite N veces.

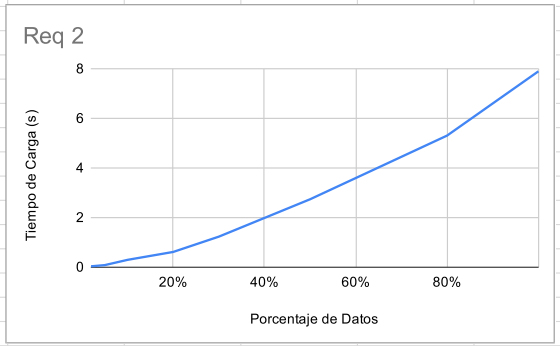
	Carga de datos		Req 1		Req 2		Req 3		Req 4		Req 5
2%	1,218	2%	0,0119	2%	0,029	2%	0,01296	2%		2%	0,004
5%	1,259	5%	0,0295	5%	0,0767	5%	0,02193	5%		5%	0,00798
10%	1,386	10%	0,0299	10%	0,2895	10%	0,0568	10%		10%	0,01790
20%	1,764	20%	0,0513	20%	0,6085	20%	0,1632	20%		20%	0,03280
30%	2,842	30%	0,069	30%	1,226	30%	0,225	30%		30%	0,04780
50%	5,392	50%	0,1105	50%	2,735	50%	0,493	50%		50%	0,10170
80%	12,298	80%	0,1774	80%	5,304	80%	1,036	80%		80%	0,21560
100%	20,092	100%	0,2508	100%	7,896	100%	1,664	100%		100%	0,51420



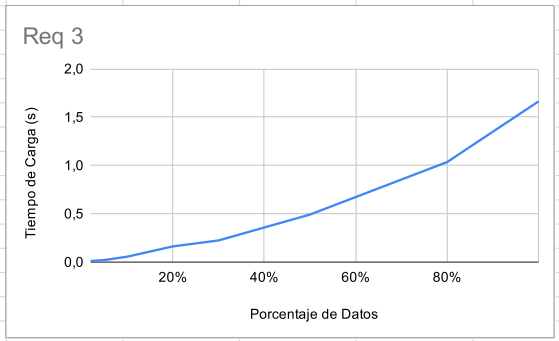
Observamos un crecimiento cuadratico de la carga de datos, lo cual se explica por la cantidad de estructuras de datos que se tienen que construir



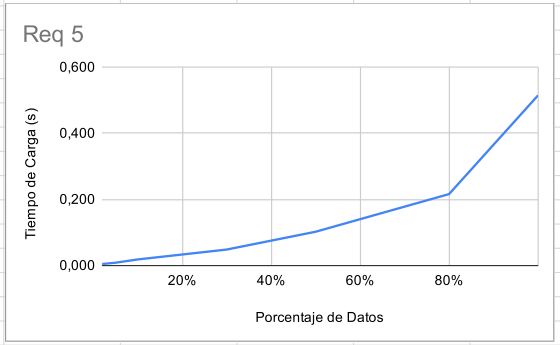
Observamos que el crecimiento temporal del requerimiento 1 es lineal, ya que la funcion de mayor complejidad para el requerimiento solo usa un iterador, es decir, su complejidad es O(v)



Observaos que el crecimiento temporal del req 2 tiene un comportamiento lineal, ya que se utiliza el algoritmo de Kosajaru que tiene complejdad O(V+E)



Obeservamos que el comportamiento del req 3 es lineal aunque podria ser semicuadratico, en especial porque la complejidad mas significativa de este algoritmo es O(h*k + k), donde k es el numero de aeropuertos encontrados en el cuadrado de busqueda y h el numero de veces que se expande el cuadrado de busqueda.



El comportamiento del tiempo de ejecucion del requerimiento 5 en relacion al numero de datos no es lineal. Dentro del requerimiento la función con mayor complejidad temporal se comporta como O(E*V), donde E es el numero de arcos y V el número de vertices. Si se mirara la mayor cantidad de arcos en un digrafro es V(V-1) lo que explica el crecimiento de tiempo en relación al número de datos.