Análisis de los resultados: Reto 4

Estudiante A: Santiago Ballén Cod 202023544

Estudiante B: Paola Campiño Cod 202020785

Aspectos preliminares:

Reto 4: https://github.com/EDA2021-1-SEC07-G-2Grupo/Reto4-S11-G02.git

	Máquina 1	**Máquina 2**
Procesadores	AMD Ryzen 5 2500U with radeon Vega Mobile Gfx, 2.00GHz	Intel Core i3-3220 CPU @ 3.30GHz
Memoria RAM (GB)	6,00GB (5,65GB utilizable)	4,00 GB (3,89 GB utilizable)
Sistema Operativo	Sistema operativo de 64 bits, procesador x64	Sistema operativo de 64 bits, procesador x64

El documento continuación presenta los siguientes datos:

- Análisis de complejidad en notación O para cada uno de los requerimientos.
- Análisis de tiempo de ejecución y uso de memoria para cada uno de los requerimientos y la carga de datos.
- Gráfico con análisis de cada uno de los requerimientos y de la carga de datos.

*Los análisis de tiempo de ejecución y el uso de memoria para los requerimientos 1,2,3,4 y 5 se hicieron en con la maquina 1.

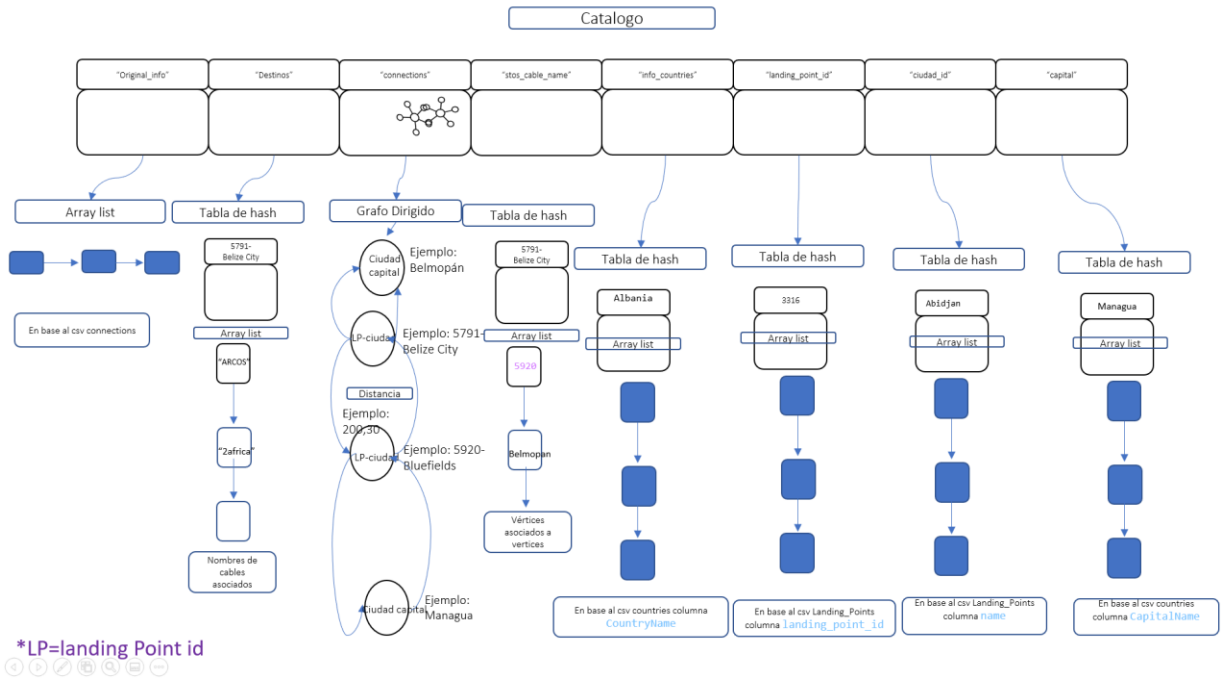
Contenido

Carga de Datos:

Tiempos de ejecución y consumo de Datos:

Consumo de Datos[kB]	Tiempo de Ejecución [ms]
39402.62	6298.06

Grafico:



Requerimiento 1: Identificación de clústeres de comunicación

Tiempos de ejecución y consumo de Datos:

Datos con los que se evaluó el requerimiento

landing_point1	landing_point2
Redondo Beach	Vung Tau

Respuesta:

La tabla a continuación presenta el tiempo que tardó el programa en cargar todos los datos en un catálogo:

Consumo de Datos[kB]	Tiempo de Ejecución [ms]
2204.16	2879.40

Complejidad del Requerimiento:

Para este primer requerimiento lo primero que se hiso fue aplicar el algoritmo de Kosaraju para encontrar todos los componentes conectados por medio de la función que se ve a continuación:

```
    analyzer['connections_scc'] = scc.KosarajuSCC(analyzer['connections'])

    return scc.connectedComponents(analyzer['connections_scc'])
```

Después de aplicarse el algoritmo lo que se está haciendo es que el retorno se está guardando en el catálogo, apara así finalmente usar la función connectedComponents que nos dirá cuantos elementos se encuentran fuertemente conectados.

La complejidad del requerimiento al aplicar KosarajuSCC es de: O (v log v)

*siendo v el número de vértices en el grafo

Y la complejidad al aplicar connectedComponents es de: O(k)

Por lo que hasta este punto el requerimiento tiene una complejidad de: O(v log v)

Lo siguiente que va a pasar es que vamos a encontrar si 2 vértices están fuertemente conectados, por lo que vamos a pasar el string del usuario a la manera en la que hemos construido nuestro grafo. De manera que va a ser necesario aplicar la función vertices_buscables:

```
def vertices_buscables(catalog,v):
    elemento=m.get(catalog["ciudad_id"],v)
    if elemento==None:
        return v
    else:

        info=lt.firstElement(elemento["value"]["song"])
        return v+"-"+str(info["landing_point_id"]
```

toda esta función tiene una complejidad de O(k), por lo que hasta este punto la complejidad total será de: O(v log v)

Aplicada la función anterior y ya transformados ambos vértices a consultar es importante aplicar la función strongly_conected que nos permitirá saber si los vértices a consultar están conectados en algún punto:

```
def strongly_conected(catalog,v1,v2):

    if m.contains(catalog["connections_scc"]["idscc"],v1)==False or m.contains
(catalog["connections_scc"]["idscc"],v2)==False:
        return False
    elif  scc.stronglyConnected(catalog["connections_scc"],v1,v2)==False:
        return 0
    elif  scc.stronglyConnected(catalog["connections_scc"],v1,v2)==True:
        return True
```

Para esto simplemente llamaremos el retorno dado por el algoritmo de kosarajo y aplicaremos la función la función stronglyConnected que como resultado nos va a dar true o false

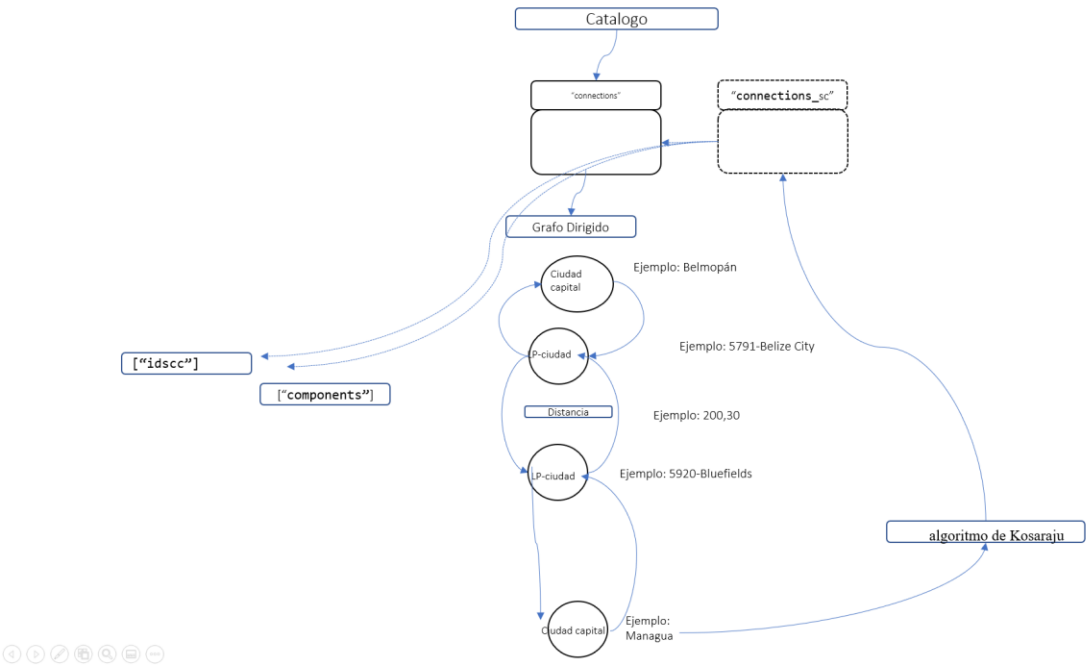
dependiendo si los elementos están conectados. Esta ultima parte tiene una complejidad de O(K)

```
def stronglyConnected(scc, verta, vertb):
    """
    Dados dos vértices, informa si están fuertemente conectados o no.
    """
    try:
        scca = map.get(scc['idscc'], verta)['value']
        sccb = map.get(scc['idscc'], vertb)['value']
        if scca == sccb:
            return True
        return False
    except Exception as exp:
        error.reraise(exp, 'dfo:Sconnected')
```

En conclusión la complejidad total del requerimiento es de: O (v log v)

- v siendo la cantidad de vértices en el grafo

Grafico:



Requerimiento 2: Identificar los puntos de conexión críticos de la red

Tiempos de ejecución y consumo de Datos:

Respuesta:

La tabla a continuación presenta el tiempo que tardó el programa en cargar todos los datos en un catálogo:

Consumo de Datos[kB]	Tiempo de Ejecución [ms]
4.83	1122.75

Complejidad del Requerimiento:

Para este requerimiento 2 fue importante que en la carga de datos se creara un diccionario en el cual la llave sería los vértices del grafo con el nombre “connections” y los valores eran todos los cables asociados a dicho landing point.

Por lo que para este requerimiento es esencial usar la tabla de hash con el nombre de “stos_cable_name”.

```
critic_list=lt.newList(datastructure="ARRAY_LIST")
    landing_point_id=m.keySet(catalog["stos_cable_name"])
    for ids in lt.iterator(landing_point_id):
        temporary_list=[]
        info =m.get(catalog["stos_cable_name"],ids)

        for cables in lt.iterator(info["value"]):
            if cables not in temporary_list:
                temporary_list.append(cables)


        if len(temporary_list)>1:
            datos=m.get(catalog["landing_point_id"],info["key"])
            if datos !=None:
                first= datos["value"]["song"]

                name=first["elements"][0]["id"]

                pais=ciudad(catalog,info["key"])
                dato={"identificador":info["key"],"Pais":pais,"name":name,"con
ectados":str(len(temporary_list)) }
                lt.addLast(critic_list,dato)
```

Lo que está pasando en la función que se presenta anteriormente es que se está haciendo un recorrido por todo la tabla de hash con el fin de ver si en algún landing point tiene más de un cable asociado. De manera que si un landing point tiene más de 1 cable con nombre diferente se va a presentar al usuario.

Para esto lo primero que se hiso fue sacar todas las llaves que están presentes en la tabla de hash, este proceso tiene una complejidad de O(n) ya que se está haciendo un recorrido en busca de las llaves.

Después lo que va a pasar es que se va a hacer un recorrido de todas las llaves en busca de todos los valores asociados a dicha llave, esto tiene una complejidad de O(n). Lo que sigue es que se va a recorrer a cada uno de los elementos asociados y se van a guardar en una lista, dicho proceso tiene una complejidad de O(e), siendo e el numero de elementos asociados a dicha llave.

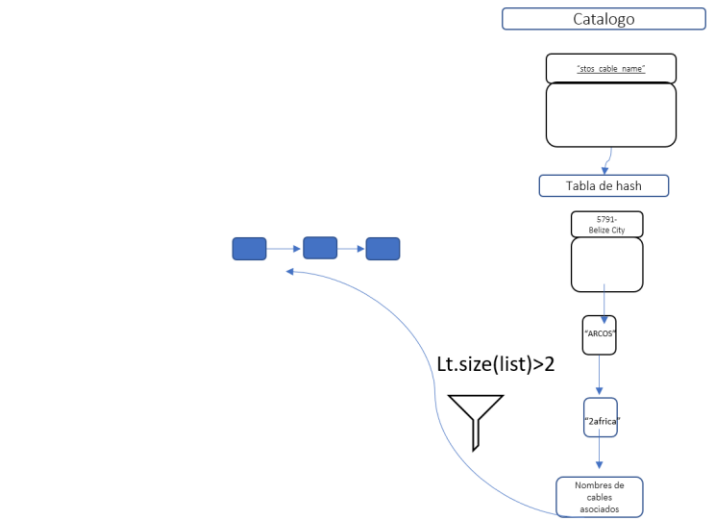
Todo esto tiene una complejidad de O(n+n*e)

- n siendo la cantidad de llaves dentro de la tabla de hash
- e siendo el numero de elementos asociados a un landing point (en gran parte de los casos e<n)

Al final la complejidad total del requerimiento es de O(n(1+e))

- n siendo la cantidad de llaves dentro de la tabla de hash
- e siendo el numero de elementos asociados a un landing point (en gran parte de los casos e<n)

Grafico:



Requerimiento 3: La ruta de menor distancia

Tiempos de ejecución y consumo de Datos:

Datos con los que se evaluó el requerimiento

País 1	País 2
Colombia	Indonesia

Respuesta:

La tabla a continuación presenta el tiempo que tardó el programa en cargar todos los datos en un catálogo:

Consumo de Datos[kB]	Tiempo de Ejecución [ms]
1589.27	1548.65

Complejidad del Requerimiento:

Para este tercer requerimiento lo primero que se hiso después de recibir los países a consultar fue necesario, usar la función getcity, que nos permitirá hacer el recorrido desde un vértice ciudad capital.

```
def getcity(catalog,pais):
    info_pais=m.get(catalog["info_countries"],pais)
    if info_pais==None:
        return None
    else:
        elemento=lt.firstElement(info_pais["value"]["song"])
        return elemento["CapitalName"]
```

Hasta este punto la complejidad de todo el requerimiento es de O(K)

Lo siguiente que va a pasar es que se va a aplicar el algoritmo de dijikstra en base a la ciudad 1 por medio de la función `Dijkstra` y se va a guardar en el catálogo con el nombre de `paths.`

```
def dijkstra_path(catalog,ciudad1):

        catalog['paths'] = djk.Dijkstra(catalog['connections'], ciudad1)
        return catalog
```

Hasta este punto la complejidad de requerimiento 3 es de: O(a log v)

- a la cantidad de arcos en el grafo
- v la cantidad de vértices en el arco

Lo siguiente que va a pasar es que se va a aplicar la función que aparece a continuación para poder sacar el camino más corto hacía el país B.

```
def dijkstra_llegada(catalog,ciudad2):
    path = djk.pathTo(catalog['paths'], ciudad2)
    return path
```

Esta función tiene una complejidad de O(n) como se muestra a continuación:

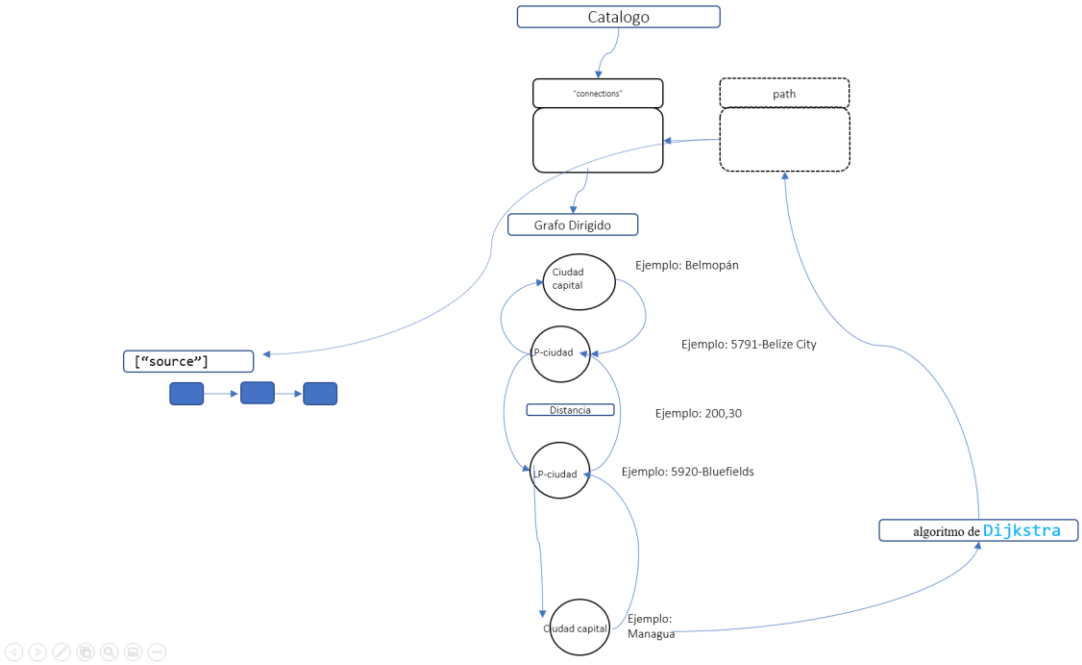
- siendo n los arcos necesarios para llegar al punto deseado

```
def pathTo(search, vertex):
    """
    Retorna el camino entre source y vertex
    en una pila.
    Args:
        search: La estructura de busqueda
        vertex: El vertice de destino
    Returns:
        Una pila con el camino entre source y vertex
    Raises:
        Exception
    """
    try:
        if hasPathTo(search, vertex) is False:
            return None
        path = stack.newStack()
        while vertex != search['source']:
            visited_v = map.get(search['visited'], vertex)['value']
            edge = visited_v['edgeTo']
            stack.push(path, edge)
            vertex = e.either(edge)
        return path
    except Exception as exp:
        error.reraise(exp, 'dks:pathto')
```

Al final este requerimiento presentara una complejidad final de: O(a log v + n)

- a la cantidad de arcos en el grafo
- v la cantidad de vértices en el arco
- siendo n los arcos necesarios para llegar al punto deseado

Grafico:



Requerimiento 4: Identificar la Infraestructura Crítica de la Red

Tiempos de ejecución y consumo de Datos:

Respuesta:

La tabla a continuación presenta el tiempo que tardó el programa en cargar todos los datos en un catálogo:

Consumo de Datos[kB]	Tiempo de Ejecución [ms]
2399.52	2612.18

Complejidad del Requerimiento:

Para este 4to requerimiento fue necesario hacer un mst, en este caso usamos el algoritmo prim(eager). Por medio de la función PrimMST la cual permite aplicar el algorithmo prim dando como respuesta un mst nos va a ser útil al momento de presentar la información necesaria. El mst va a ser guardado en el catalogo con el nombre “prim”.

```
def prim_search(catalog):
```

```
    catalog["prim"] = prim.PrimMST(catalog)
    return catalog["prim"]
```

En si el algoritmo prim tiene una complejidad de: O(n^2)

*n siendo la cantidad de arcos

Como el requerimiento pide la cantidad de nodos conectados a la ruta de expansión mínima el siguiente paso es hacer uso de la función m.size sobre la tabla de hash con el nombre [“marked”] como se muestra a continuación:

```
total_nodos=model.nodos_totales(prime["marked"])


def nodos_totales(mst):
    return m.size(mst)
```

Hacer uso de esta función tiene una complejidad de O(k)

El siguiente paso es encontrar la conexión más larga después de hacer el mst, para esto se va a usar la función mergesort de manera que después de sacar el mst podamos organizar los elementos en base al costo necesario para llegar a cada elemento. A continuación, se puede ver la función usada para sacar este dato:

```
def ruta_min(mst):
    mst=mst["distTo"]
    keys=m.keySet(mst)
    suma=0.0
    for elementos in lt.iterator(keys):
        info = m.get(mst,elementos)

        suma+=info["value"]
    return suma
```

Lo que se está haciendo es que se está sumando todos los costos desde una tabla de hash que contiene toda la información en cuando a costo para llegar a cada uno de los vértices.

Hasta este punto la complejidad total es de: O(n^2+n)

*n siendo la cantidad de vértices

Y para finalizar este requerimiento es importante llegar a saber cual es vértice que se encuentra alejado y cual es el vértice más cercano. Para encontrar esta información es importante hacer uso de una función que permita organizar los elementos en base al valor.

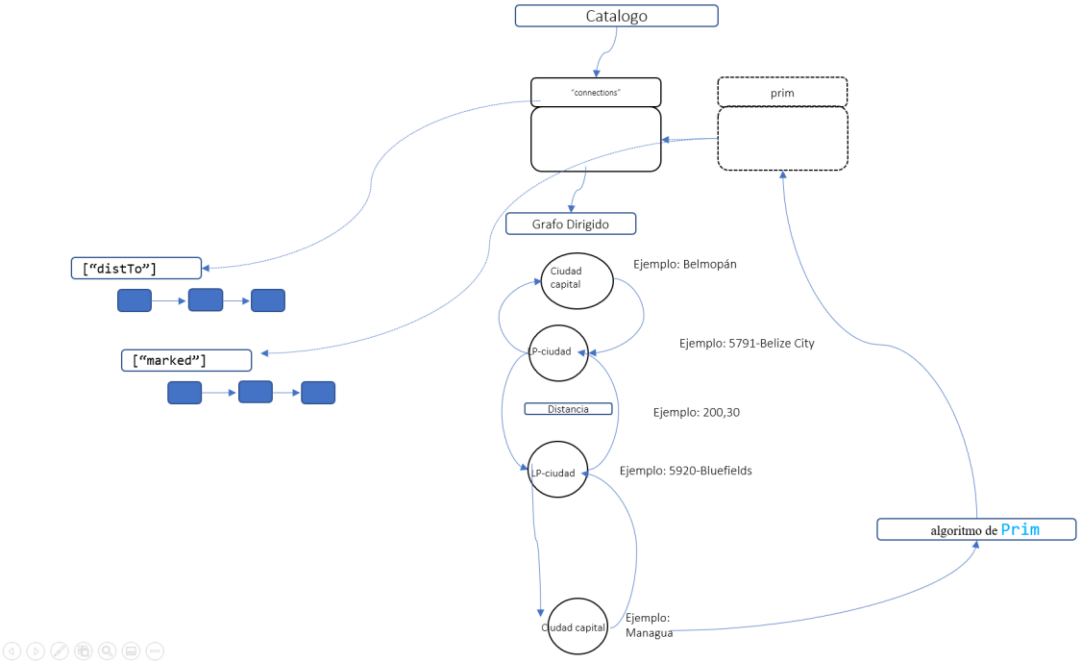
```
def merge_sort(lista,size,compare_funct):
    sub_list = lt.subList(lista,0, size)
    sub_list = lista.copy()
```

```
    sorted_list=merg.sort(sub_list, compare_funct)
    return  sorted_list
```

En este caso voy a usar el algorithmo de ordenamiento mergesort, ya que en retos pasados ha resultado muy rápido y eficiente al momento de organizarlos elementos. Sin embargo esto hace que la complejidad sea un más alta, dando como resultado un requerimiento con una complejidad de: O(n^2+n+2log n)

*n es la cantidad de arcos en el grafo original

Grafico:



Requerimiento 5: Análisis de fallas

Tiempos de ejecución y consumo de Datos:

Landing Point
Fortaleza

La tabla a continuación presenta el tiempo que tardó el programa en cargar todos los datos en un catálogo:

Consumo de Datos[kB]	Tiempo de Ejecución [ms]
0.17	28.72

Complejidad del Requerimiento:

Para este quinto requerimiento se uso la función gr.adjacent de manera que fuese posible obtener los vértices que inmediatamente tendrían problemas.

```
        element = map.get(graph['vertices'], vertex)
        lst = element['value']
        lstresp = lt.newList()
        for edge in lt.iterator(lst):
            v = e.either(edge)
            if (v == vertex):
                lt.addLast(lstresp, e.other(edge, v))
            else:
                lt.addLast(lstresp, v)
        return lstresp
    except Exception as exp:
        error.reraise(exp, 'ajlist:adjacents')
```

Hasta este punto la complejidad total del requerimiento es de: O(N)

- n siendo el número de vértices en el peor caso

Lo siguiente a hacer es consultar el país relacionado a cada uno de los landing points previamente obtenidos, por lo que vamos a hacer uso de la siguiente función:

```
def landing_paises(catalog,lista,vertice):
    newlist=lt.newList(datastructure="ARRAY_LIST")

    lst_element=[]
    for element in lt.iterator(lista):
        distancia=gr.getEdge(catalog["connections"],element,vertice)
        ciudad=des_vertice(element)
        lista_grande=m.get(catalog["ciudad_id"],ciudad)
        if lista_grande!=None:
            dato=lt.firstElement(lista_grande["value"]["song"])
            pais=dato["name"].split(",")
            if len(pais)>2:
                if  pais[2] not in lst_element:
                    info={"Pais":str(pais[2]),"Distancia": str(round(float(dis
tancia["weight"]),2))}
                    lt.addLast(newlist,info)
                    lst_element.append(pais[2])
            elif len(pais)==2:
                if pais[1] not in lst_element:
                    info={"Pais":str(pais[1]),"Distancia": str(round(float(dis
tancia["weight"]),2))}
                    lt.addLast(newlist,info)
                    lst_element.append(pais[1])
```

```
            else:
                if pais[0] not in lst_element:
                    info={"Pais":str(pais[0]),"Distancia": str(round(float(dis
tancia["weight"]),2))}
                    lt.addLast(newlist,info)
                    lst_element.append(pais[0])

    return newlist
```

La cual nos permite consultar las diferentes tablas de hash con el fin de crear una nueva lista con solo la información necesarias

Ya siendo este el final del requerimiento es correcto afirmar que la complejidad final es de: O(2N)

- siendo n el numero de vértices presentes en el grafo

Grafico: