ANÁLISIS DEL RETO

Estudiante 1: Javier Steven Barrera Toro, 202214779, js.barrerat1@uniandes.edu.co

Estudiante 2: Sergio Andres González Mateus, 202210910, sa.gonzalezm123@uniandes.edu.co

Estudiante 3 Harel Daniel Neira Galindo, 201816386, h.neira@uniandes.edu.co

Requerimiento <<1>>

```python
# Requerimiento 1
def routeStations(catalog, station_i, station_f):
    gr_stations = catalog["gr_stations"]
    paths = djk.Dijkstra(gr_stations, station_i)
    lt_stations_t = lt.newList("ARRAY_LIST")
    steps = []
    stations_t, tranfers_t = 0,0

    if djk.hasPathTo(paths, station_f):
        path = djk.pathTo(paths, station_f)

        while not st.isEmpty(path):
            step = st.pop(path)
            # se agraga el paso a la salida
            steps.append({
                "Estacion inicial": step["vertexA"],
                "Siguiente estacion": step["vertexB"],
                "Distancia": step["weight"]
                })

            if not lt.isPresent(lt_stations_t, step["vertexA"]):
                lt.addLast(lt_stations_t, step["vertexA"])
            if not lt.isPresent(lt_stations_t, step["vertexB"]):
                lt.addLast(lt_stations_t, step["vertexB"])

        distance = djk.distTo(paths, station_f)

        stations_t, tranfers_t = countStationTranfers(lt_stations_t)

    return distance, station_f, stations_t, tranfers_t, steps
```

Descripción

Breve descripción de como abordaron la implementación del requerimiento

Entrada		Identificador de la estación origen y destino
Salidas		La distancia total que tomara el camino de origen y destino, total de estaciones, total de transbordos, identificador de estación y la distancia a la siguiente estación.

Implementado (Sí/No)		Si (Grupal)

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Dijkstra y luego se organiza en una lista.	O(E+V*log(V))
Identifica el camino al vértice de destino	O(V)
Cuenta los vértices	O(V)
TOTAL	**O(E+V*log(V))**

Requerimiento <<2>>

```python
def fewStops(catalog, station_i, station_f):
    digr_stops = catalog["digr_stops"]
    paths = djk.Dijkstra(digr_stops, station_i)
    mp_stations = catalog["mp_stations"]
    lt_stations_t = lt.newList("ARRAY_LIST")
    steps = []
    distance_t = 0
    stations_t, tranfers_t = 0,0

    if djk.hasPathTo(paths, station_f):
        path = djk.pathTo(paths, station_f)

        while not st.isEmpty(path):
            step = st.pop(path)
            v_station_i = me.getValue(mp.get(mp_stations, step["vertexA"]))
            v_station_f = me.getValue(mp.get(mp_stations, step["vertexB"]))
            lon_1, lat_1 = float(v_station_i["Longitude"]), float(v_station_i["Latitude"])
            lon_2, lat_2 = float(v_station_f["Longitude"]), float(v_station_f["Latitude"])
            distance = harvesineDistance(lon_1, lat_1, lon_2, lat_2)
            # se agraga el paso a la salida
            steps.append({
                "Estacion inicial": step["vertexA"],
                "Siguiente estacion": step["vertexB"],
                "Distancia": distance
                })

            if not lt.isPresent(lt_stations_t, step["vertexA"]):
                lt.addLast(lt_stations_t, step["vertexA"])
            if not lt.isPresent(lt_stations_t, step["vertexB"]):
                lt.addLast(lt_stations_t, step["vertexB"])

            distance_t += distance

        # realiza un conteo de las estaciones y transbordos
        stations_t, tranfers_t = countStationTranfers(lt_stations_t)

    return distance_t, stations_t, tranfers_t, steps      You, 9 hours ago • add: req 1
```

Descripción

Breve descripción de como abordaron la implementación del requerimiento

Entrada	Se desea conocer el camino más corto entre una estación de origen y otra de destino.

Salidas	La distancia total que tomará el camino entre la estación de origen y estación destino, total de estaciones que contiene el camino, total de transbordos de tuta que debe realizar el usuario, el identificador de la estación y la distancia a la siguiente estación.
Implementado (Sí/No)	Si (Grupal)

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Dijkstra y luego se organiza en una lista.	O(E+V*log(V))
Identifica el camino al vértice de destino	O(V)
Cuenta los vértices	O(V)
TOTAL	**O(E+V*log(V))**

Requerimiento <<3>>

```
# Requerimiento 3
def component(catalog):
    catalog['mp_stations'] = scc.KosarajuSCC(catalog['gr_stations'])
    return scc.connectedComponents(catalog['mp_stations'])
```

Descripción

Breve descripción de como abordaron la implementación del requerimiento

Entrada	No se requiere parámetros de entrada para el requerimiento.
Salidas	El total de componentes conectados dentro del grafo, mostrar 5 componentes conectados más grandes, el número de estaciones que pertenece a ese componente y identificar las 3 primeras y últimas estaciones que pertenecen al componente.
Implementado (Sí/No)	Si (Individual) Harel Daniel Neira

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Recorre el grafo por KosarajuSCC y devuelve los componentes conectados de este.	O(N)
Paso 2	O(...)
Paso ….	O(...)
TOTAL	***O(N)***

Requerimiento <<4>>

```python
# Requerimiento 4
def planShortRoute(catalog, lon_i, lat_i, lon_f, lat_f):
    gr_stations = catalog["gr_stations"]
    mp_stations = catalog["mp_stations"]
    lt_vertices = catalog["lt_vertices"]
    lt_station_i = lt.newList("ARRAY_LIST")
    lt_station_f = lt.newList("ARRAY_LIST")
    lt_stations_t = lt.newList("ARRAY_LIST")
    steps = []
    distance_t = 0
    stations_t = 0
    tranfers_t = 0

    # recorre cada vertice del grafo
    for station in lt.iterator(lt_vertices):
        # si la estacion no es un transbordo calcula la distancia del las estaciones de entreda a las estaciones de los vertices
        if station[0:1] != "T":
            v_station = me.getValue(mp.get(mp_stations, station))
            lon_s, lat_s = float(v_station["Longitude"]), float(v_station["Latitude"])

            distance_i = harvesineDistance(lon_i, lat_i, lon_s, lat_s)
            lt.addLast(lt_station_i, [distance_i, station])
            distance_f = harvesineDistance(lon_f, lat_f, lon_s, lat_s)
            lt.addLast(lt_station_f, [distance_f, station])

    # ordena de menor a mayor las estaciones en funcion de la distancia
    shs.sort(lt_station_i, compareDistance)
    shs.sort(lt_station_f, compareDistance)

    near_stationi = lt.getElement(lt_station_i, 1)
    near_stationf = lt.getElement(lt_station_f, 1)
    station_i = near_stationi[1]
    station_f = near_stationf[1]
    # calcula el camino mas corto desde un vertice de inicio
    paths = djk.Dijkstra(gr_stations, station_i)

    # si existe el camino al vertice busca el camino al vertice final
    if djk.hasPathTo(paths, station_f):
        path = djk.pathTo(paths, station_f)
        # mientras la pila del camino no este vacia se elimina un elemento
        while not st.isEmpty(path):
            step = st.pop(path)
            # se agraga el paso a la salida
            steps.append({
                "Estacion inicial": step["vertexA"],
                "Siguiente estacion": step["vertexB"],
                "Distancia": step["weight"]
                })

            # se agregan los vertices que se visitaron
            if not lt.isPresent(lt_stations_t, step["vertexA"]):
                lt.addLast(lt_stations_t, step["vertexA"])
            if not lt.isPresent(lt_stations_t, step["vertexB"]):
                lt.addLast(lt_stations_t, step["vertexB"])
        # encuentra la distancia desde el vertice inicial al vertice final
        distance_t = djk.distTo(paths, station_f)

    # realiza un conteo de las estaciones y transbordos
    stations_t, tranfers_t = countStationTranfers(lt_stations_t)

    return station_i.split("-")[0], near_stationi[0], station_f.split("-")[0], near_stationf[0], distance_t, stations_t, tranfers_t, steps
```

Descripción

Breve descripción de como abordaron la implementación del requerimiento

Entrada	Localización geográfica de origen y destino por latitud y longitud del usuario.
Salidas	La distancia entre localización de origen y la estación de bus más cercana, la distancia total del recorrido entre la estación origen y destino, la distancia entre la estación destino más cercana y la localización destino, el total de estaciones, el total de transbordos, la identificación de la estación y la distancia a la siguiente estación en el camino.
Implementado (Sí/No)	Si (Individual) Sergio González

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Recorre cada vértice en la lista de vértices; N en este caso son todos los vértices.	O(N)
Sort de lt_station_i y lt_station_f	O(N^2)
Paso ….	O(...)
TOTAL	***O(N^2)***

Requerimiento <<5>>

```python
# Requerimiento 5
def reachable_stations(catalog: dict, id_org: str, num_max: int):
    graph_digr_stops = catalog['digr_stops']
    mp_stations = catalog['mp_stations']

    paths = djk.Dijkstra(graph_digr_stops, id_org)
    vertices = gr.vertices(graph_digr_stops)
    lst = lt.newList('ARRAY_LIST')
    format_list = []

    for vertex in lt.iterator(vertices):
        dist_to = djk.distTo(paths, vertex)
        if djk.hasPathTo(paths, vertex) and (dist_to <= num_max) and (dist_to != 0):
            lt.addLast(lst, vertex)

    for vertex in lt.iterator(lst):
        longitud = djk.distTo(paths, vertex)
        v_station_i = me.getValue(mp.get(mp_stations, vertex))
        lon_1, lat_1 = float(v_station_i["Longitude"]), float(v_station_i["Latitude"])
        format_list.append({
            "station": vertex,
            "latitude": lat_1,
            "longitude": lon_1,
            "steps": longitud
        })

    return format_list
```

Descripción

Breve descripción de como abordaron la implementación del requerimiento

Entrada	Identificador de la estación de origen y número de conexiones permitidas desde la estación de origen.
Salidas	Información de las estaciones “alcanzables” con el número de conexiones, identificador de cada estación, geolocalización de la estación por latitud y longitud, por último, longitud del camino desde la estación de origen y estación alcanzada.
Implementado (Sí/No)	Si (Individual) Javier Barrera

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1	O(...)
Paso 2	O(...)
Paso ….	O(...)
TOTAL	***O(...)***

Requerimiento <<6>>

```python
# Requrimiento 6
def routeStationNH(catalog, station_i, nh):
    gr_stations = catalog["gr_stations"]
    mp_nh = catalog["mp_nh"]
    lt_dist_stations = lt.newList("ARRAY_LIST")
    paths = djk.Dijkstra(gr_stations, station_i)
    lt_nh = me.getValue(mp.get(mp_nh, nh))

    for stop in lt.iterator(lt_nh):
        station = stop["Code"]
        bus = stop["Bus_Stop"].replace(" ", "").split("-")[1]
        code_ruta = station + "-" + bus
        if djk.hasPathTo(paths, code_ruta) and (code_ruta != station_i) and (code_ruta[0:1] != "T"):
            dist = djk.distTo(paths, code_ruta)
            lt.addLast(lt_dist_stations, [dist, code_ruta])

    shs.sort(lt_dist_stations, compareDistance)
    station_f = lt.getElement(lt_dist_stations, 1)[1]

    return routeStations(catalog, station_i, station_f)
```

Descripción

Breve descripción de como abordaron la implementación del requerimiento

Entrada	Identificador de estación de origen y el identificador del vecindario.
Salidas	Distancia total que tomara el recorrido entre la estación de origen y destino del vecindario destino, total de estación que contiene el camino, total de transbordos de ruta, camino calculado entre las estaciones de origen y destino junto con su identificador de la estación, el identificador del vecindario la distancia a la siguiente estación.
Implementado (Sí/No)	Si (Grupal)

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Dijkstra	O(E+V*log(V))

Recorrer estaciones de vecindarios y agregarlos a la lista	O(V)
Ordenamiento de la lista por Shell sort	O (V ^2)
Dijkstra y luego se organiza en una lista.	O(E+V*log(V))
Identifica el camino al vértice de destino	O(V)
TOTAL	**O (V ^2)**

Requerimiento <<7>>

```python
# Requerimiento 7
def circularPath(catalog, station_i):
    gr_stations = catalog["gr_stations"]
    lt_vertices = catalog["lt_vertices"]
    lt_path_i = lt.newList("ARRAY_LIST")
    lt_path_f = lt.newList("ARRAY_LIST")
    paths_i = djk.Dijkstra(gr_stations, station_i)
    steps_i = []
    steps_f = []
    find = False
    tranfers_t = 0
    stations_t = 0
    distance = 0

    for station_f in lt.iterator(lt_vertices):
        if djk.hasPathTo(paths_i, station_f) and (not find) and station_i != station_f and station_f[0:1] != "T":
            paths_f = djk.Dijkstra(gr_stations, station_f)
            if djk.hasPathTo(paths_f, station_i):
                find = True
                path_i = djk.pathTo(paths_i, station_f)
                path_f = djk.pathTo(paths_f, station_i)
                distance = djk.distTo(paths_i, station_f) + djk.distTo(paths_f, station_i)

                values = [{"path": path_i, "steps": steps_i, "lt_path": lt_path_i}, {"path": path_f, "steps": steps_f, "lt_path": lt_path_f}]

                for i in range(0,2):
                    while not st.isEmpty(values[i]["path"]) and find == True:
                        step = st.pop(values[i]["path"])
                        values[i]["steps"].append({
                                "Estacion inicial": step["vertexA"],
                                "Siguiente estacion": step["vertexB"],
                                "Distancia": step["weight"]
                                })

                        if not lt.isPresent(values[i]["lt_path"], step["vertexA"]):
                            lt.addLast(values[i]["lt_path"], step["vertexA"])
                        if not lt.isPresent(values[i]["lt_path"], step["vertexB"]):
                            lt.addLast(values[i]["lt_path"], step["vertexB"])

                # realiza un conteo de las estaciones y transbordos
                for station in lt.iterator(lt_path_i):
                    if station[0:1] == "T":
                        tranfers_t += 1
                    else:
                        stations_t += 1

                for station in lt.iterator(lt_path_f):
                    if station[0:1] == "T":
                        tranfers_t += 1
                    else:
                        stations_t += 1

    return distance, stations_t, tranfers_t, steps_i, steps_f
```

Descripción

Breve descripción de como abordaron la implementación del requerimiento

Entrada	Identificador de la estación de origen.
Salidas	La distancia total del recorrido del camino circular está siendo mayor a 0.0, el total de estaciones que contiene el camino (debe ser mayor a 1), el total transbordos, el camino calculado entre las

	estaciones de origen y destino de la siguiente manera: El identificador de la estación y la distancia a la siguiente estación.
Implementado (Sí/No)	Si (Grupal)

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Dijkstra	O(E+V*log(V))
Recorrer los vértices	O(V)
Formatear la lista	O(V)
Contar las estaciones	O(V)
TOTAL	**O(E+V*log(V))**

Tablas de datos

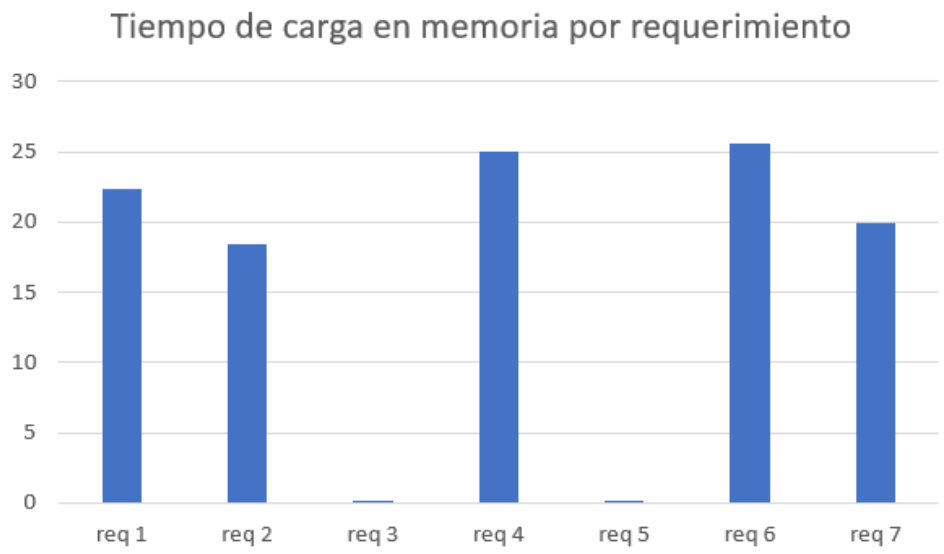
Las tablas con la recopilación de datos de las pruebas.

Graficas

Las gráficas con la representación de las pruebas realizadas.

Req 1	Req 2	Req 3	Req 4	Req 5	Req 6	Req 7
2923,74	36,223	0	205,78	0	5076,09	2822,96

req 1	req 2	req 3	req 4	req 5	req 6	req 7
22,375	18,46	0,17	25,02	0,17	25,64	19,89


Tiempo de carga en memoria por requerimiento
30
25
20
15
10
5
0
req 1
req 2
req 3
req 4
req 5
req 6
req 7