ANÁLISIS DEL RETO

Tomas Aponte, 202420148, t.aponte@uniandes.edu.co

Juan Diego García, 202423575, jd.garcia12@uniandes.edu.co

Carga de datos

```python
def load_data(catalog, filename=(data_dir + "deliverytime_min.csv")):
    """
    Carga los datos del reto
    """
    # TODO: Realizar la carga de datos
    file = open(filename, encoding="utf-8")
    restaurants = al.new_list()
    domicilios = al.new_list()
    rappis = al.new_list()
    reader = csv.DictReader(file)
    t_total = 0
    t_counter = 0
    for row in reader:
        delivery ={
            "ID": row["ID"],
            "Delivery_person_ID": row["Delivery_person_ID"],
            "Delivery_person_Age": row["Delivery_person_Age"],
            "Delivery_person_Ratings": row["Delivery_person_Ratings"],
            "Restaurant_latitude": row["Restaurant_latitude"],
            "Restaurant_longitude": row["Restaurant_longitude"],
            "Delivery_location_latitude": row["Delivery_location_latitude"],
            "Delivery_location_longitude": row["Delivery_location_longitude"],
            "Type_of_order": row["Type_of_order"],
            "Type_of_vehicle": row["Type_of_vehicle"],
            "Time_taken": int(row["Time_taken(min)"]),
        }
        al.add_last(catalog["registros"], delivery)
        if row["Delivery_person_ID"] not in rappis:
            al.add_last(rappis, row["Delivery_person_ID"])
        r_loctation = (row["Restaurant_latitude"]+"_"+row["Restaurant_longitude"])
        d_location = (row["Delivery_location_latitude"]+"_"+row["Delivery_location_longitude"])
        if r_loctation not in restaurants:
            al.add_last(restaurants, row["Restaurant_latitude"])
        if d_location not in domicilios:
            al.add_last(domicilios, row["Delivery_location_latitude"])
        t_total += int(row["Time_taken(min)"])
        t_counter += 1
    t_avg = t_total / t_counter
    file.close()
    return al.size(catalog["registros"]), al.size(rappis), al.size(restaurants), al.size(domicilios),  t_avg
```

Descripción

La carga de datos está dividida en 4 funciones: new_logic(), load_data(). diagraph() y conexiones_domicilios(). New_logic() se encarga de generar el catálogo donde se cargaran los registros y el grafo de los registros, dentro de un diccionario de llaves “registros” y “grafo”. Load_data() se encarga de llenar la lista de “registros” que tiene toda la información asociada a cada registro del archivo csv. La

lista se llena con diccionarios, donde las llaves de estos diccionarios representan las columnas del CSV y cada diccionario es una fila (o registro) del CSV. Diagraph() se encarga de comenzar a poblar el grafo, con la información cargada en registros, utilizando un ciclo for donde se generan 2 vértices por registro: uno para el restaurante y otro para el destino del domicilio; para mantener la estructura deseada de las llaves, acceder a las posiciones geográficas dentro de "registros" y las recorta a 4 decimales con la función decimales(), y así genera 2 nodos que cumplen con el formato deseado. Una vez se tenga las llaves, se verifica si el nodo ya existe para evitar duplicados, y así mantener el valor de cada nodo estable, ya que se trata de una lista que contiene a todos los "rappis" (ID's de domiciliarios) que compartan una misma ubicación geográfica. Además genera de inmediato la conexión restaurante-domicilio y domicilio-restaurante, verificando cuando debe promediar tiempos. Por último, conexiones_domicilios() se encarga de verificar que domicilios debe estar conectados si comparten el mismo "rappi" (ID de domiciliario), mediante un ciclo for que recorre parcialmente la lista de registros, y buscando coincidencias de rappis para generar las conexiones en los mapas de adyacencia.

Entrada	Catálogo, archivo
Salidas	Valores del grafo de datos cargados
Implementado (Sí/No)	Si, Juan Diego García

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Añade dato por dato a la estructura	O(n)
Verifica datos repetidos	O(1)
Genera las conexiones	O(n**2)
TOTAL	***O(n**2)***

Pruebas Realizadas

Ryzen 7 7730U

16GB

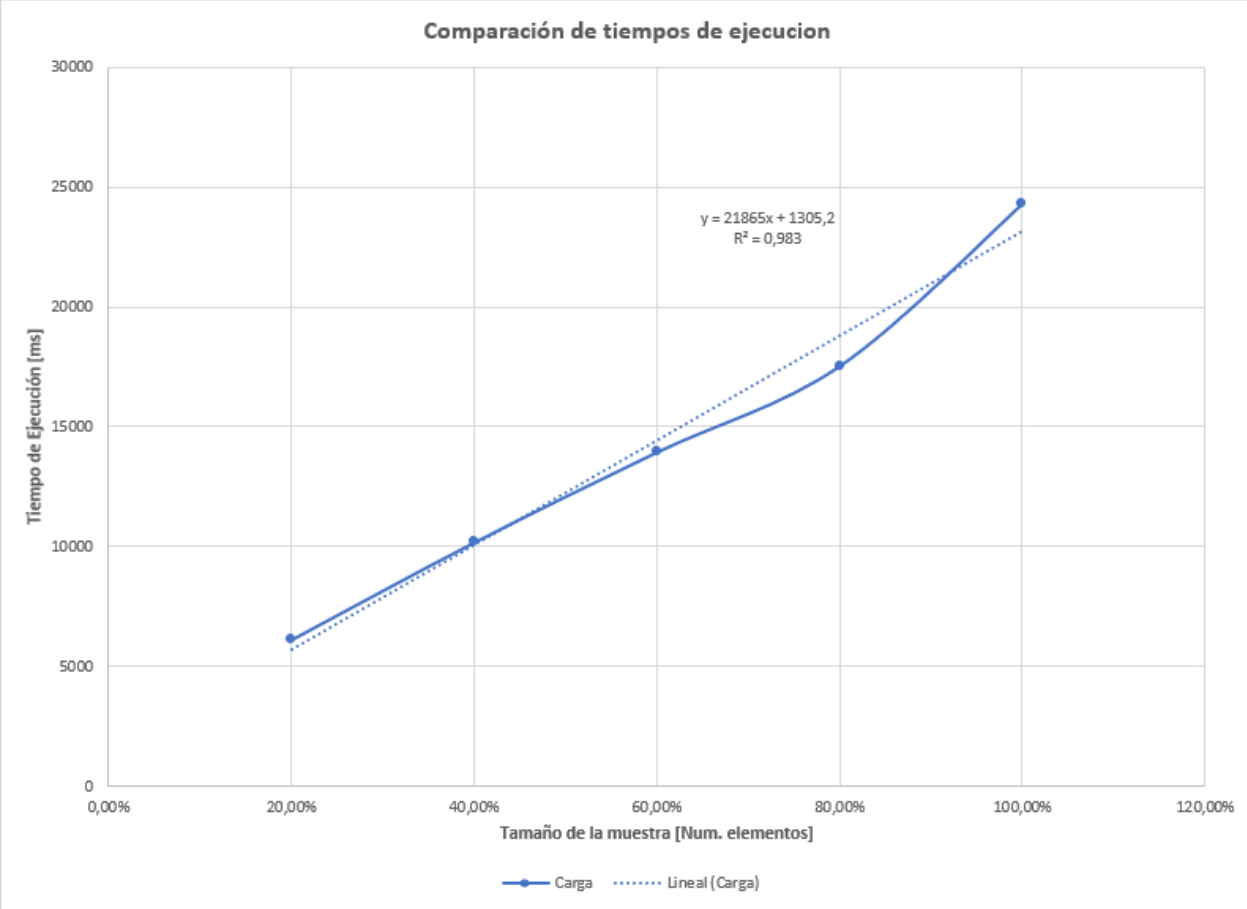
Windows 11

Entrada	**Tiempo (s)**
20%	6125.54
40%	10187.52
60%	13954.90
80%	17543.44
100%	24312.87

Tablas de datos

Porcentaje de la muestra [pct]	Tamaño de la muestra	Carga
20,00%	9118,80	6125,54
40,00%	18237,60	10187,52
60,00%	27356,40	13954,9
80,00%	36475,20	17543,44
100,00%	45594,00	24312,87

Graficas



Análisis

El comportamiento observado en los tiempos de carga fue cercano a lineal, diferente a como se esperaba, complejidad O(n**2). Cada registro se podria procesar mas de una vez y luego se inserta en estructuras con acceso constante. La mayor parte del tiempo en la carga se concentra en la construcción del grafo, especialmente en la verificación de arcos duplicados y la actualización de pesos promedio por lo que la complejidad podria llegar a ser cuadratica.

Requerimiento <<1>>

```python
def req_1(catalog, id_a, id_b):
    """
    Retorna el resultado del requerimiento 1
    """
    # TODO: Modificar el requerimiento 1
    start_time = get_time()
    conections = dfs.dfs(catalog["grafo"], id_a)
    path = dfs.path_to(id_b, conections)
    if path is None:
        end_time = get_time()
        time = delta_time(start_time, end_time)
        return None, time
    else:
        restaurants = al.new_list()
        node = path["first"]
        ids = al.new_list()
        while node is not None:
            i = 0
            centinela = False
            while not centinela:
                if catalog["registros"]["elements"][i]["ID"] == node["info"]:
                    lat_1 = decimales(catalog["registros"]["elements"][i]["Restaurant_location_latitude"])
                    long_1 = decimales(catalog["registros"]["elements"][i]["Restaurant_location_longitude"])
                    key = (lat_1 + "_" + long_1)
                    if mp.contains(conections["marked"], key):
                        al.add_last(restaurants, key)
                    if catalog["registros"]["elements"][i]["Delivery_person_ID"] not in ids:
                        al.add_last(ids, catalog["registros"]["elements"][i]["Delivery_person_ID"])
                        centinela = True
                i += 1
            node = node["next"]
        end_time = get_time()
        time = delta_time(start_time, end_time)
        return ids, path["size"], path["elements"], restaurants, time
```

Descripción

El requerimiento identifica un camino simple entre dos distintos IDs de ubicaciones. A partir de un recorrido DFS se crea un camino para el cual se verifican las longitudes de los arcos que lo componen, los elementos y los restaurantes visitados para luego añadirlos a una lista y retornarlos.

Entrada	Catálogo, ID_A, ID_B
Salidas	Cantidad de puntos, ID de los domiciliarios encontrados, secuencia del camino y listado de restaurantes.
Implementado (Sí/No)	Si, Juan Diego García

Análisis de complejidad

Pasos	**Complejidad**
Recorrido DFS	O(V+E)
Añadir elementos a la lista	O(n)
TOTAL	***O(V+E)***

Pruebas Realizadas

Ryzen 7 7730U

16GB

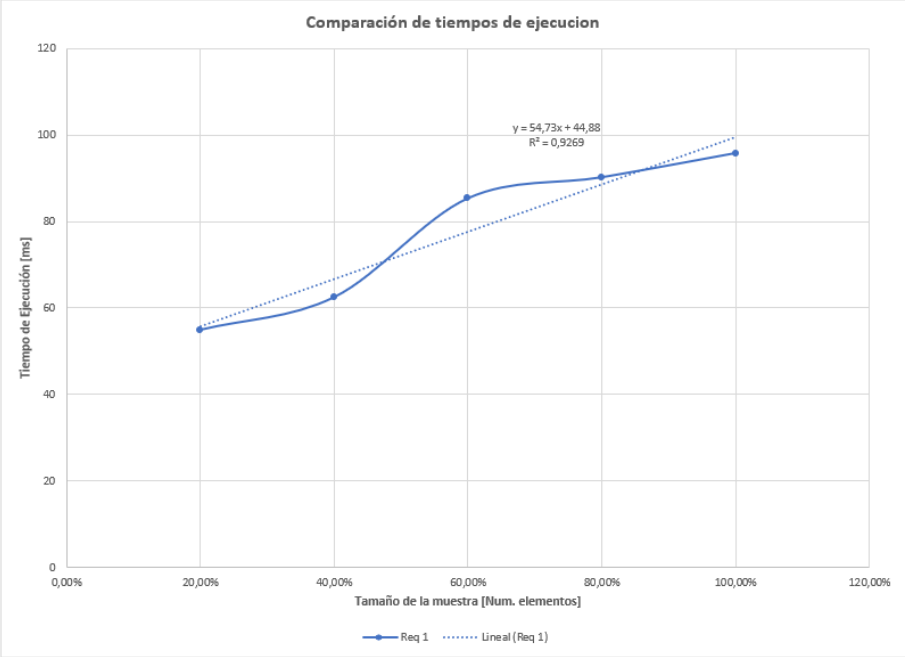
Windows 11

Entrada	Tiempo (s)
20	54,89
40	62,45
60	85,32
80	90,17
100	95,76

Tablas de datos

Porcentaje de la muestra [pct]	Tamaño de la muestra	Req 1
20,00%	9118,80	54,89
40,00%	18237,60	62,45
60,00%	27356,40	85,32
80,00%	36475,20	90,17
100,00%	45594,00	95,76

Graficas



Análisis

El comportamiento observado coincide con la teoría, el DFS escala bien con el tamaño del grafo y no presenta subidas repentinas en tiempo. Las diferencias pequeñas pueden atribuirse a la carga de procesamiento adicional al reconstruir el camino o recolectar datos como restaurantes y domiciliarios.

Requerimiento <<3>>

```python
def req_3(catalog, id):
    """
    Retorna el resultado del requerimiento 3
    """
    # TODO: Modificar el requerimiento 3
    start_time = get_time()
    centinela = False
    j = 0
    while not centinela and j < al.size(catalog["registros"]):
        if catalog["registros"]["elements"][j]["ID"] == id:
            lat = catalog["registros"]["elements"][j]["Restaurant_latitude"]
            long = catalog["registros"]["elements"][j]["Restaurant_longitude"]
            centinela = True
        else:
            j += 1
    if not centinela:
        end_time = get_time()
        time = delta_time(start_time, end_time)
        return None, time
    apariciones = {}
    vehiculos = {}
    llave = decimales(lat) + "_" + decimales(long)
    rappis = dg.get_vertex_information(catalog["grafo"], llave)["value"]
    for i in range(al.size(catalog["registros"])):
        if catalog["registros"]["elements"][i]["Restaurant_latitude"] == lat and catalog["registros"]["elements"][i]["Restaurant_longitude"] == long:
            if catalog["registros"]["elements"][i]["Delivery_person_ID"] in rappis:
                if catalog["registros"]["elements"][i]["Delivery_person_ID"] not in apariciones:
                    apariciones[catalog["registros"]["elements"][i]["Delivery_person_ID"]] = 1
                else:
                    apariciones[catalog["registros"]["elements"][i]["Delivery_person_ID"]] += 1
                if catalog["registros"]["elements"][i]["Delivery_person_ID"] not in vehiculos:
                    vehiculos[catalog["registros"]["elements"][i]["Delivery_person_ID"]] = al.new_list()
                    al.add_last(vehiculos[catalog["registros"]["elements"][i]["Delivery_person_ID"]], catalog["registros"]["elements"][i]["Type_of_vehicle"])
                else:
                    al.add_last(vehiculos[catalog["registros"]["elements"][i]["Delivery_person_ID"]], catalog["registros"]["elements"][i]["Type_of_vehicle"])
    if apariciones:
        rappi = max(apariciones, key=apariciones.get)
        delivery_count = apariciones[rappi]
        vehicles = vehiculos[rappi]
        contador_2 = {}
        for vehicle in vehicles:
            if vehicle not in contador_2:
                contador_2[vehicle] = 1
            else:
                contador_2[vehicle] += 1
        vehicle = max(contador_2, key=contador_2.get)
        preferencia_vehiculo = contador_2[vehicle]
    end_time = get_time()
    time = delta_time(start_time, end_time)
    return rappi, delivery_count, preferencia_vehiculo, time
```

Este requerimiento busca identificar el domiciliario con mayor cantidad de pedidos en un punto geográfico determinado. Se recorre la lista de registros almacenados en el catálogo, filtrando aquellos domicilios que tengan como origen o destino el punto consultado, y se contabiliza la frecuencia de cada domiciliario. Finalmente, se retorna el domiciliario más frecuente, la cantidad de veces que aparece y su tipo de vehículo más usado.

Descripción

Entrada	Catálogo, ID
Salidas	Domiciliario mas solicitado, cantidad de pedidos, tipo de vehículo
Implementado (Sí/No)	Si, Juan Diego García

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Verifica la existencia del domiciliario	O(n)
Verifica la cantidad de apariciones del domiciliario	O(n+k)
Verifica el vehiculo mas utilizado	O(n)
TOTAL	***O(n+k)***

Pruebas Realizadas

Ryzen 7 7730U

16GB

Windows 11

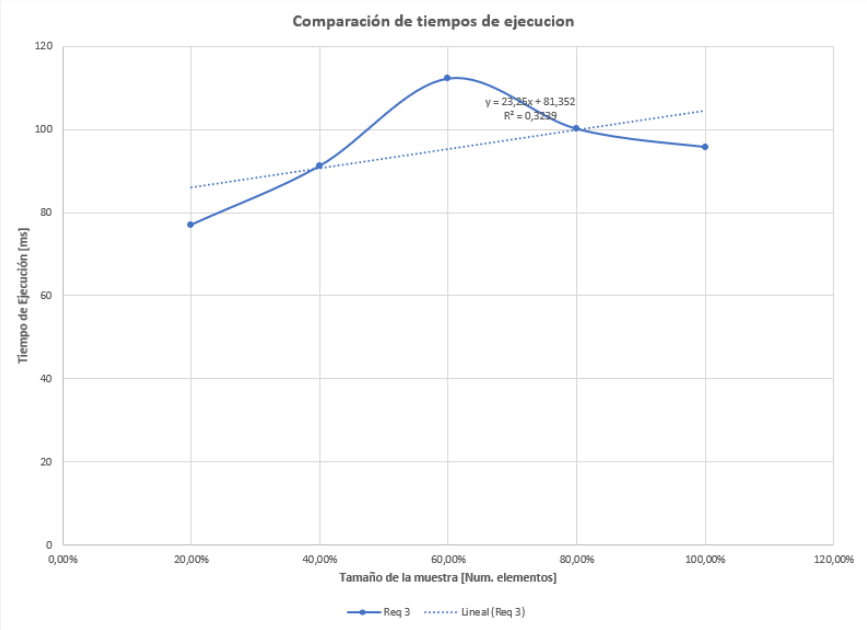
ID: DCD4

Entrada	**Tiempo (s)**
20	76,98
40	91,23
60	112,37
80	100,17
100	95,76

Tablas de datos

Porcentaje de la muestra [pct]	Tamaño de la muestra	Req 3
20,00%	9118,80	76,98
40,00%	18237,60	91,23
60,00%	27356,40	112,37
80,00%	36475,20	100,17
100,00%	45594,00	95,76

Graficas



Análisis

El algoritmo fue eficiente y la curva de tiempos sigue parcialmente la curva esperada. Hay un levantamiento grande en la curva al procesar el 60% de los datos, esto es un comportamiento inesperado que podria atribuirse a alguna variacion en las estructuras utilizadas para almacenar la informacion.

Requerimiento <<4>>

```python
def req_4(catalog, punto_A, punto_B):
    """
    Retorna el resultado del requerimiento 4
    """
    start_time = get_time()

    graph = catalog["grafo"]

    bfs_result = bfs.bfs(graph, punto_A)

    path = bfs.path_to(bfs_result, punto_B)

    if not path:
        return 0.0, [], []

    secuencia_ubicaciones = al.new_list()
    domiciliarios_por_punto = al.new_list()

    for punto in path:
        al.add_last(secuencia_ubicaciones,(punto))
        vertex_info = dg.get_vertex_information(graph, punto)
        ids_en_punto = al.new_list()
        al.add_last(ids_en_punto,vertex_info["value"]["elements"])
        al.add_last(domiciliarios_por_punto,(ids_en_punto))

    domiciliarios_comunes = al.new_list()
    al.add_last(domiciliarios_comunes,domiciliarios_por_punto)

    end_time = get_time()
    time = delta_time(start_time, end_time)

    return time, secuencia_ubicaciones, domiciliarios_comunes
```

Este requerimiento identifica los domiciliarios en común que aparecen en el camino simple con menor número de puntos intermedios entre dos ubicaciones A y B. Para esto se realiza un recorrido BFS desde A hasta B. Se extraen los domiciliarios asociados a cada ubicación del camino, y se calcula la intersección entre esas listas usando únicamente listas array_list.

Descripción

Entrada	Catálogo, punto A, punto B
Salidas	Secuencia de ubicaciones y lista de domiciliarios comunes.
Implementado (Sí/No)	Si, Tomas Aponte

Análisis de complejidad

Pasos	**Complejidad**
Realizar recorrido BFS	O(V+E)
Guardar el camino	O(n)
Añade los datos de el recorrido en diferentes listas	O(n)

TOTAL	***O(V+E)***

Pruebas Realizadas

Ryzen 7 7730U

16GB

Windows 11

A: 1CF1

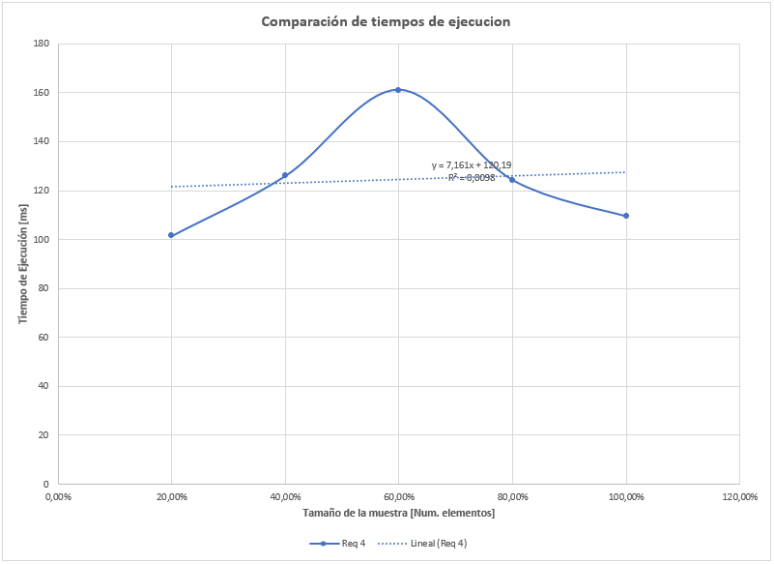
B: 4a51

Entrada	Tiempo (s)
20	101,54
40	125,98
60	161,09
80	124,302
100	109,54

Tablas de datos

Porcentaje de la muestra [pct]	Tamaño de la muestra	Req 4
20,00%	9118,80	101,54
40,00%	18237,60	125,98
60,00%	27356,40	161,09
80,00%	36475,20	124,302
100,00%	45594,00	109,54

Graficas



Análisis

El recorrido se comporta cierta medida como se esperaba. Aunque, la complejidad al cargar lel 60% de los datos varia en una muy gran medida. Puede atribuirse a la creacion de caminos diferentes por el BFS. En general se podria decir que a gran escala la curva deberia acercarse a un orden lineal.

Requerimiento <<6>>

```python
def req_6(catalog, id):
    """
    Retorna el resultado del requerimiento 6
    """
    # TODO: Modificar el requerimiento 6
    start_time = get_time()
    dijkstra_result = dij.dijkstra(catalog["grafo"], id)
    visited = dijkstra_result["visited"]
    reachable = {}
    keys = mp.key_set(visited)
    for key in keys:
        if mp.get(visited, key) < float('inf'):
            reachable[key] = mp.get(visited, key)
    sorted_reachable = sorted(reachable.keys())
    max_vertex = max(reachable, key=reachable.get)
    max_cost = reachable[max_vertex]
    path = reconstruct_path(catalog["grafo"], visited, id, max_vertex)
    end_time = get_time()
    time = delta_time(start_time, end_time)

    return path, path["size"], sorted_reachable, path["elements"], max_cost, time
```

Este requerimiento busca encontrar todos los caminos de costo mínimo en tiempo desde una ubicación geográfica inicial A a todas las demás ubicaciones alcanzables. Se implementa el algoritmo de Dijkstra, calculando los pesos mínimos desde el nodo origen. Se retorna el número de nodos alcanzados, el tiempo total acumulado y el camino más costoso entre ellos.

Descripción

Entrada	Catálogo, ID
Salidas	Camino mas corto, su tamaño, orden, elementos y costo
Implementado (Sí/No)	Si, Juan Diego García

Análisis de complejidad

Pasos	**Complejidad**
Dijkstra	O(V+E LogV)
Identificar alcanzable	O(n)
Reconstruir camino	O(E log V)
TOTAL	***O(V+E LogV)***

Pruebas Realizadas

Ryzen 7 7730U

16GB

Windows 11

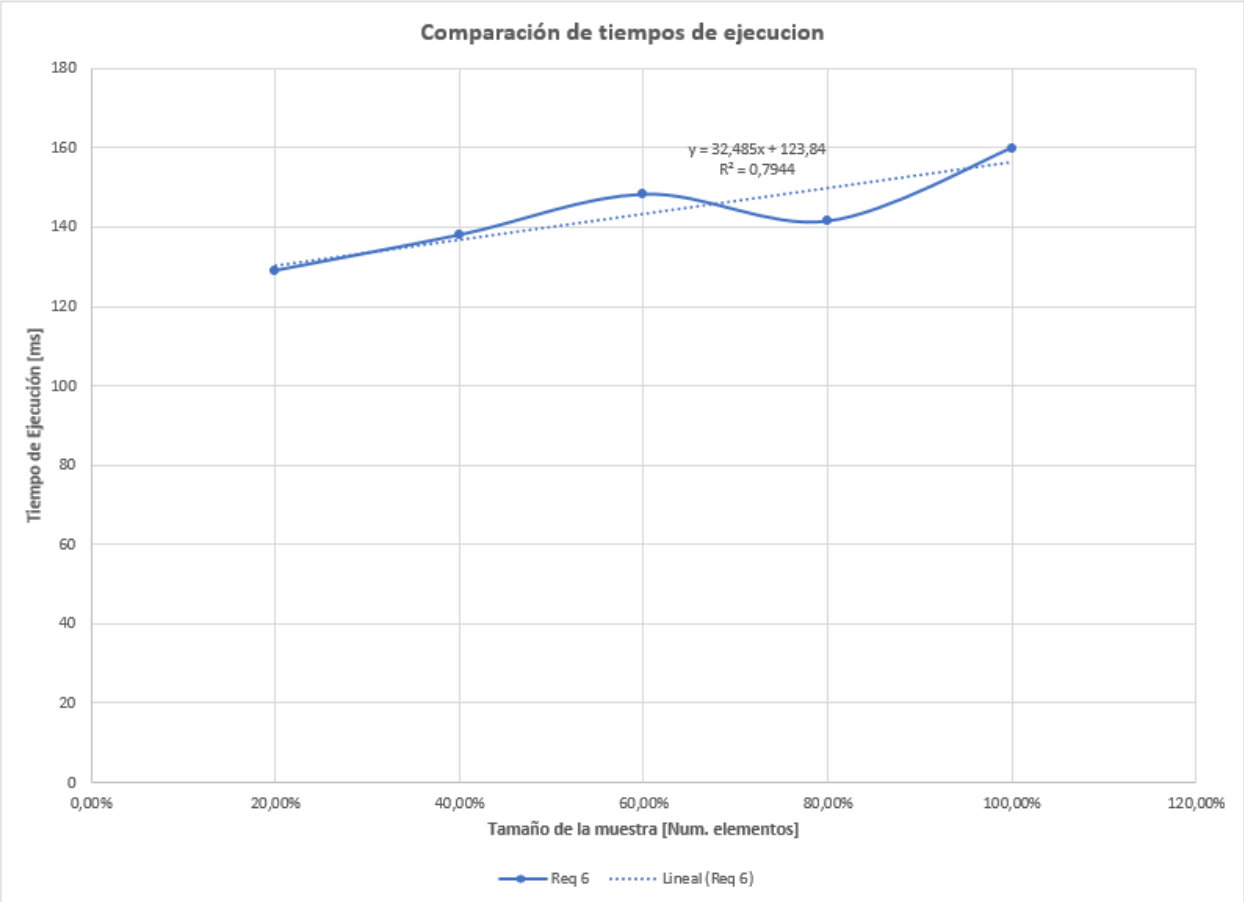
ID: B54C

Entrada	Tiempo (s)
20	129,09
40	138,02
60	148,23
80	141,47
100	159,85

Tablas de datos

Porcentaje de la muestra [pct]	Tamaño de la muestra	Req 6
20,00%	9118,80	129,09
40,00%	18237,60	138,02
60,00%	27356,40	148,23
80,00%	36475,20	141,47
100,00%	45594,00	159,85

Graficas



Análisis

El rendimiento coincide con lo esperado teóricamente, gracias a una implementación correcta de Dijkstra. La eficiencia se mantuvo constante tambien con los archivos más grandes, lo que demuestra que el algoritmo fue bien adaptado a la estructura del grafo del reto y no hay resultados erraticos que puedan indicar algun error al ejecutar el algoritmo.

Requerimiento <<7>>

```python
def req_7(catalog, punto_A, domiciliario_id):
    """
    Retorna el resultado del requerimiento 7
    """
    # TODO: Modificar el requerimiento 7
    start = get_time()
    grafo_original = catalog["grafo"]

    subgrafo = dg.new_graph(1000)
    vertices = grafo_original["vertices"]["table"]

    for entry in vertices:
        if entry is not None:
            id_nodo = entry["key"]
            lista_domiciliarios = entry["value"]
            if al.is_present(lista_domiciliarios, domiciliario_id):

                dg.insert_vertex(subgrafo, id_nodo, lista_domiciliarios)

    for entry in vertices:
        if entry is not None:
            u = entry["key"]
            if dg.contains_vertex(subgrafo, u):
                adyacentes = dg.adjacents(grafo_original, u)
                for i in range(1, al.size(adyacentes)+1):
                    v = al.get_element(adyacentes, i)
                    if dg.contains_vertex(subgrafo, v):
                        arco = dg.get_edge(grafo_original, u, v)
                        if not dg.contains_vertex(subgrafo, u, v):
                            dg.add_edge(subgrafo, u, v, arco["weight"])


    mst_result = prim.prim(subgrafo, punto_A)


    total_peso = 0
    ubicaciones = al.new_list()

    edges = mst_result["edges"]
    for i in range(1, al.size(edges)+1):
        edge = al.get_element(edges, i)
        total_peso += edge["weight"]
        if not al.is_present(ubicaciones, edge["vertexA"]):
            al.add_last(ubicaciones, edge["vertexA"])
        if not al.is_present(ubicaciones, edge["vertexB"]):
            al.add_last(ubicaciones, edge["vertexB"])

    ubicaciones_py = al.new_list()
    for i in range(1, al.size(ubicaciones)+1):
        al.add_last(ubicaciones_py,al.get_element(ubicaciones, i))
    al.merge_sort(ubicaciones_py)

    stop = get_time()
    tiempo_total = delta_time(start,stop)

    return {
        "tiempo_ejecucion": tiempo_total,
        "num_ubicaciones": al.size(ubicaciones_py),
        "ubicaciones": ubicaciones_py,
        "costo_total": round(total_peso, 2)
    }
```

El requerimiento construye una subred desde la ubicación inicial A, considerando solo las ubicaciones por las que ha pasado un domiciliario específico. Se genera un subgrafo filtrado y se ejecuta el algoritmo de Prim desde el nodo A, restringiendo los nodos a aquellos visitados por el domiciliario. Se retorna el tiempo de ejecución, el total de nodos en la subred, la lista ordenada alfabéticamente de las ubicaciones, y el costo total del árbol.

Descripción

Entrada	Catalogo, Punto A, ID
Salidas	Ubicaciones, numero de ubicaciones, costo total
Implementado (Sí/No)	Si, Tomas Aponte

Análisis de complejidad

Pasos	Complejidad
Filtrar nodos	O(V)
Copiar arcos	O(E)
Prim	O(V log V+E)
Ordenar ubicaciones	O(V log V)
TOTAL	***O(V log V+E)***

Pruebas Realizadas

Ryzen 7 7730U

16GB

Windows 11

A: 2BED

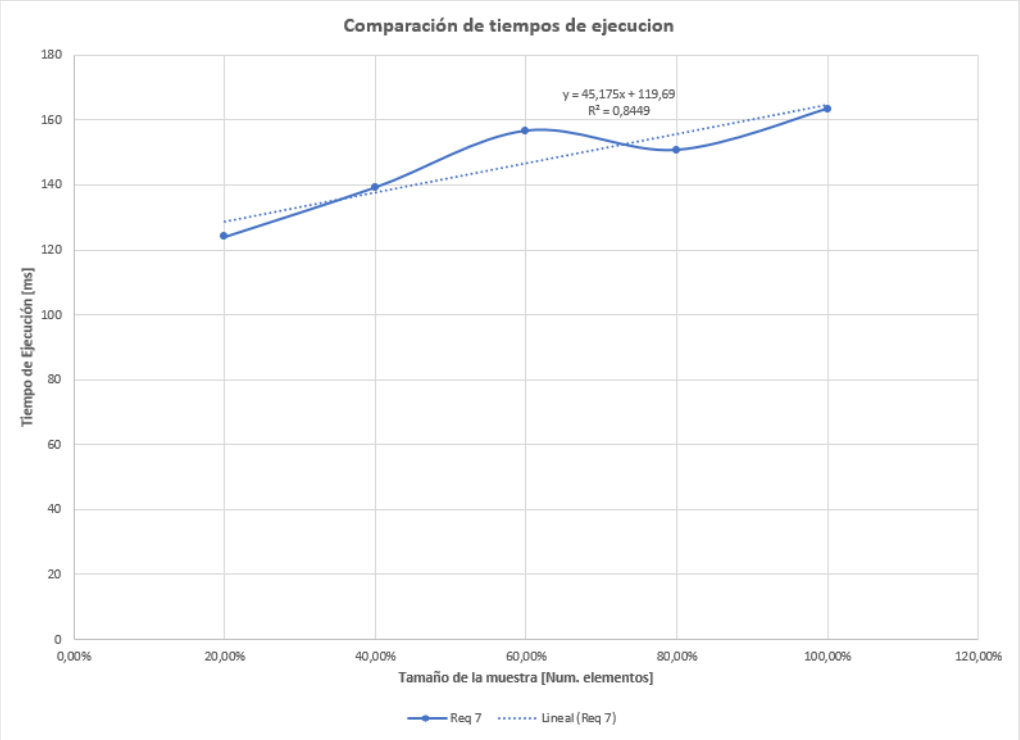
ID: INDORES13DEL02

Entrada	Tiempo (s)
20	123,98
40	139,21
60	156,65
80	150,74
100	163,39

Tablas de datos

Porcentaje de la muestra [pct]	Tamaño de la muestra	Req 7
20,00%	9118,80	123,98
40,00%	18237,60	139,21
60,00%	27356,40	156,65
80,00%	36475,20	150,74
100,00%	45594,00	163,39

Graficas



Análisis

El algoritmo de Prim fue eficiente al limitar el subgrafo a los nodos del domiciliario, reduciendo significativamente el tamaño de datos a tomar en cuenta. El proceso de ordenamiento de ubicaciones para la salida también fue lineal respecto al tamaño reducido del subgrafo. El rendimiento fue estable y consistente incluso con conjuntos de datos de gran tamaño.