ANÁLISIS DEL RETO

Andrés Felipe Rodriguez Acosta, af.rodrigueza12345@uniandes.edu.co, 202322292
Juan David Gutierrez Rodriguez, jd.gutierrezr123@uniandes.edu.co, 202316163
Samuel Rodriguez Robledo, s.rodriguezr234567@uniandes.edu.co, 202323878

Introducción

En este reto nuestro objetivo es aplicar los conocimientos aprendidos durante el módulo cuatro del curso Estructuras de datos y algoritmos. Nos centraremos en la implementación de grafos, y distintos algoritmos de búsqueda como DFS (Deepth First Search), el algoritmo de Dijkstra, entre otros. En este documento llevaremos a cabo un análisis de la complejidad de los algoritmos empleados y su costo de ejecución.

Funciones

Requerimiento 3: *Juan David Gutierrez Rodriguez*

Requerimiento 4: *Andrés Felipe Rodriguez Acosta*

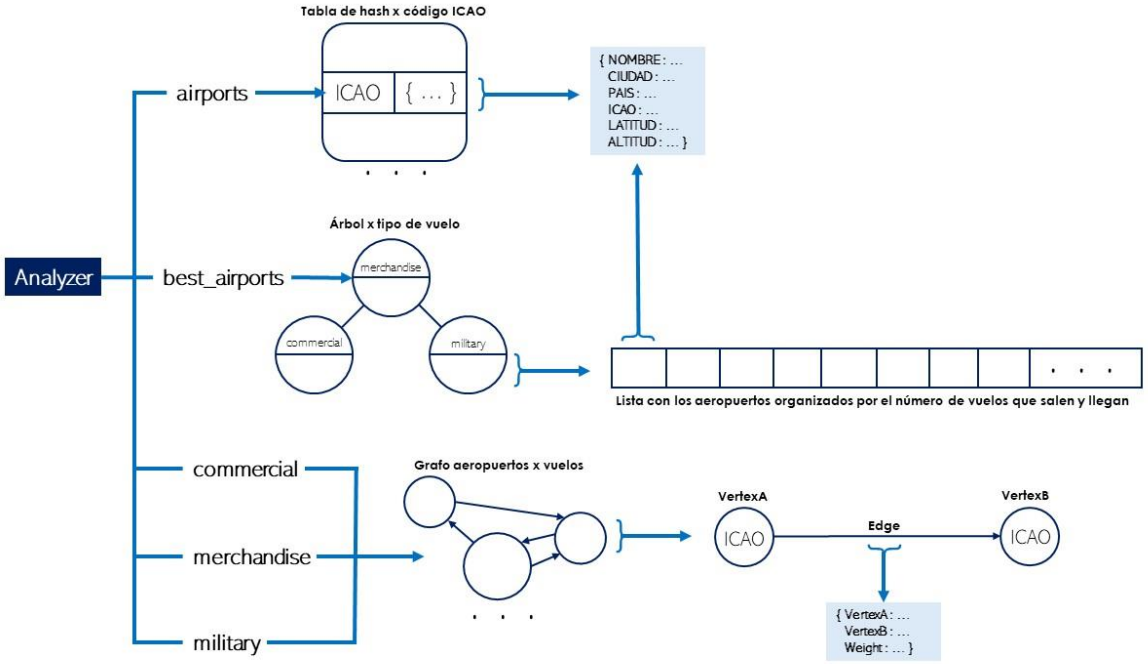
Requerimiento 5: *Samuel Rodriguez Robledo*

Índice

Carga de datos

Descripción

Para el manejo de nuestra carga de datos, hemos optado por implementar nuestras estructuras de datos acorde con el siguiente diagrama.



Inicialmente, hemos definido la estructura de nuestro catálogo, que incluirá una tabla de hash de los aeropuertos. Un árbol con tres llaves, cada una con una lista de los aeropuertos organizados por su concurrencia dependiendo el tipo de vuelo. Y finalmente, un grafo dirigido por cada tipo de vuelo el cual tendrá como vértices los aeropuertos y como arcos los vuelos. El peso de cada arco puede ser el tiempo de vuelo, o la distancia entre los aeropuertos usando la fórmula de Harvesine.

```
# Tabla de hash para almacenar los aeropuertos
analyzer['airports'] = mp.newMap(numelements=430,
                                 maptype='CHAINING',
                                 loadfactor=5)
# Árbol que contiene los mejores aeropuertos según el tipo de vuelo
analyzer['best_airports'] = om.newMap(omaptype='RBT',
                                      cmpfunction=compareTreeKeys)
# Grafo que contiene los vuelos comerciales
analyzer['commercial'] = gr.newGraph(datastructure='ADJ_LIST',
                                     directed=True,
                                     size=100,
                                     cmpfunction=compareGraphKeys)
# Grafo que contiene los vuelos de carga
analyzer['merchandise'] = gr.newGraph(datastructure='ADJ_LIST',
                                     directed=True,
                                     size=100,
                                     cmpfunction=compareGraphKeys)
# Grafo que contiene los vuelos militares
analyzer['military'] = gr.newGraph(datastructure='ADJ_LIST',
                                     directed=True,
                                     size=100,
                                     cmpfunction=compareGraphKeys)
```

Procedemos a agregar las aeropuertos a la tabla de hash. Para esto hacemos un cambio en las coordenadas para poder implementar la formula harvesine de ser necesario. Para agregarla a la tabla de hash usamos como llave el código ICAO del aeropuerto.

```
data['LATITUD'] = data['LATITUD'].replace(',', '.')
data['LONGITUD'] = data['LONGITUD'].replace(',', '.')
data['ALTITUD'] = data['ALTITUD'].replace(',', '.')

add_map(analyzer['airports'], mp, data['ICAO'], data)
```

Luego, armamos los grafos para cada tipo de vuelo. Para esto obtenemos el aeropuerto de origen y de destino. Verificamos el tipo de vuelo y el tipo de peso para los arcos (tiempo o distancia). Y por último, añadimos los vértices, que serán el código ICAO de cada aeropuerto, y el arco que los une.

```
# Obtener el aeropuerto de origen de la tabla de hash
origin_airport = me.getValue(get_entry(analyzer['airports'], mp, data['ORIGEN']))
# Obtener el aeropuerto de destino de la tabla de hash
destination_airport = me.getValue(get_entry(analyzer['airports'], mp, data['DESTINO']))

if weight == 'time':
    weight = int(data['TIEMPO_VUELO'])
else:
    weight = haversine(origin_airport, destination_airport)

if type == 'AVIACION_COMERCIAL':
    flights = analyzer['commercial']

elif type == 'AVIACION_CARGA':
    flights = analyzer['merchandise']

else:
    flights = analyzer['military']

# Añadir los vertices correspondientes a los aeropuertos
add_vertex(flights, origin_airport['ICAO'])
add_vertex(flights, destination_airport['ICAO'])

# Añadir arco entre los dos aeropuertos
add_edge(flights, origin_airport['ICAO'], destination_airport['ICAO'], weight)
```

Finalmente, para las listas ordenadas por concurrencia de cada tipo de vuelo, agregamos una llave a los diccionarios de los aeropuertos que sea la suma de los arcos que salen y llegan a ese vértice. Agregamos los aeropuertos a una lista y la ordenamos, de mayor a menor, por la concurrencia.

```
    # Agregar la concurrencia (grado) del aeropuerto (vertice)
    airport_flights['CONCURRENCIA'] = gr.indegree(analyzer[type], ICAO) + gr.outdegree(analyzer[type], ICAO)
    # Añadir a la lista de mejores aeropuertos
    lt.addLast(airports_lst, airport_flights)

# Ordenar los aeropuertos, de mayor a menor, por la concurrencia
sort(airports_lst, sort_airports)
```

Entrada	Inicialización de la estructura de datos
Salidas	Número de vuelos y aeropuertos cargados, los aeropuertos, separados por tipo de vuelo, ordenados por concurrencia.
Implementado (Sí/No)	Si fue implementado por Juan David.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	Complejidad
Inicializar catalogo	O(1)
Recorrer cada oferta del archivo	O(n)
Recorrer cada llave de la oferta	O(18)
Agregar a la lista, el árbol y la tabla	O(1)
Recorrer los demás archivos	O(n)
Añadir a la lista	O(1)
Obtener valor de la tabla de hash	O(1)
Total	O(n)

Pruebas Realizadas

Las pruebas fueron realizadas midiendo memoria. El algoritmo de ordenamiento para la lista de ofertas fue Merge Sort. La máquina utilizada cuenta con las siguientes capacidades.

Procesadores	11th Gen Intel® Core™ i5
Memoria RAM (GB)	8 GB
Sistema operativo	Windows 11 Home

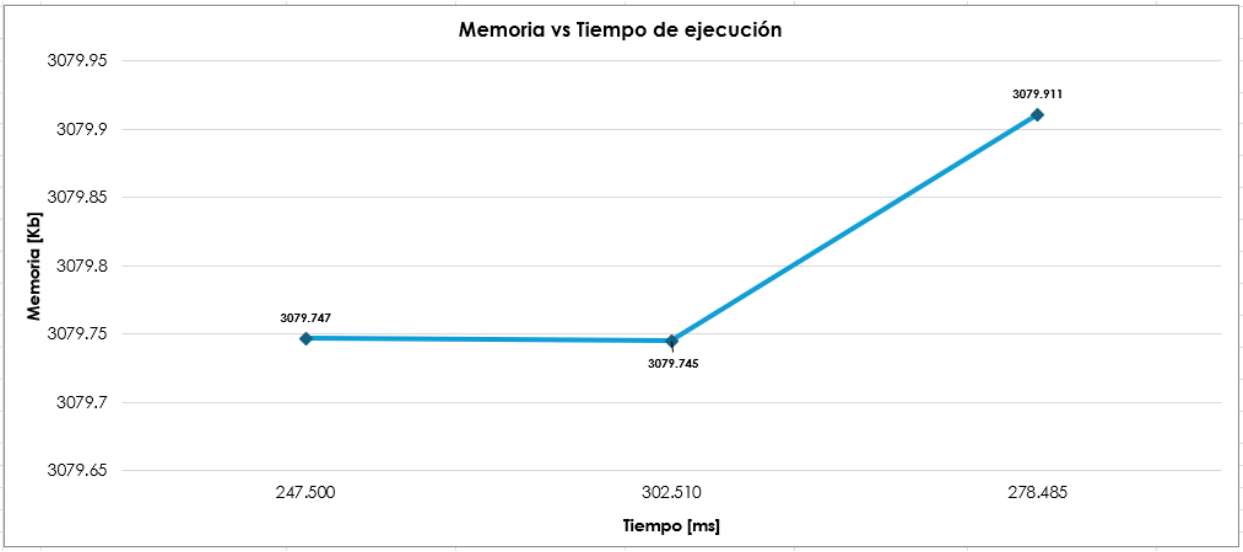
Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Aeropuertos	Vuelos			Memoria [Kb]	Tiempo [ms]
	Comerciales	Carga	Militares		
428	1195	976	849	3079.747	247.500
428	1195	976	849	3079.745	302.510
428	1195	976	849	3079.911	278.485

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

Las estructuras de datos implementadas para gestionar la carga de información relacionada con vuelos y aeropuertos están diseñadas para minimizar la complejidad, lo cual se traduce en un proceso de carga extremadamente rápido y eficiente. Gracias a esta optimización, los datos se pueden ingresar de manera ágil y con un uso mínimo de recursos de memoria, garantizando así un desempeño óptimo del sistema. Esta eficiencia no solo acelera la carga inicial de la información, sino que también contribuye a un funcionamiento más fluido y confiable durante la operación continua de los requerimientos.

Volver al índice

Requerimiento 1

Descripción

```python
def req_1(analyzer, origin, destination, map):
    """
    Función que soluciona el requerimiento 1
    """
    commercial = analyzer["commercial"]
    airports = analyzer["airports"]
    best_airports = me.getValue(get_entry(analyzer['best_airports'], om, 'commercial'))
    origin_airport = None, 100000
    destination_airport = None, 100000

    for airport in lt.iterator(best_airports):
        # Hallar el aeropuerto mas cercano a las coordenadas
        distance = haversine(airport, origin)
        if distance < origin_airport[1]:
            origin_airport = airport, distance

        distance = haversine(airport, destination)
        if distance < destination_airport[1]:
            destination_airport = airport, distance

    if origin_airport[1] > 30 or destination_airport[1] > 30:
        return (origin_airport[1], destination_airport[1]), (origin_airport[0], destination_airport[0]), None

    estructura = djk.Dijkstra(commercial,origin_airport[0]["ICAO"])
    path = djk.pathTo(estructura,destination_airport[0]["ICAO"])

    # Verificar si hay un camino entre los dos aeropuertos
    if not djk.hasPathTo(estructura, destination_airport[0]['ICAO']):
        return None, (origin_airport[0], destination_airport[0]), None
    else:
        # Hallar el camino hasta el aeropuerto
        path = djk.pathTo(estructura, destination_airport[0]['ICAO'])
```

```
flights_queue = qu.newQueue()
total_distance = origin_airport[1] + destination_airport[1]
total_time = djk.distTo(estructura, destination_airport[0]['ICAO'])
total_airports = data_size(path, st) + 1

# Ejecucion del requerimiento 8
req_8_lst = lt.newList()
lt.addLast(req_8_lst, origin_airport[0])
lt.addLast(req_8_lst, destination_airport[0])

while not st.isEmpty(path):
    flight = st.pop(path)

    airport_1 = me.getValue(get_entry(airports, mp, flight['vertexA']))
    airport_2 = me.getValue(get_entry(airports, mp, flight['vertexB']))
    # Sumar la distancia del trayecto a la suma total de distancias
    total_distance += haversine(airport_1, airport_2)

    # Agregar aeropuerto intermedio a la cola
    if airport_1['ICAO'] != origin_airport[0]['ICAO']:
        qu.enqueue(flights_queue, airport_1)

# Ejecucion del requerimiento 8
if map:
    req_8(req_8_lst)

return flights_queue,(origin_airport[0],destination_airport[0]),(total_distance,total_airports,total_time)
```

En este requerimiento el usuario debía ingresar a partir de coordenadas un punto de origen y uno de destino. Posterior a esto el requerimiento debía mediante el algoritmo Haversine, aproximar estas dos coordenadas al aeropuerto más cercano a mínimo 30 km de distancia. Este algoritmo fue implementado como una función secundaria y se desarrolló el código para que recorriera la información de todos los vuelos a través del algoritmo de Dijkstra (por preferencia del desarrollador) y que de esta manera la distancia de las coordenadas de cada vuelo se comparó a las ingresadas por parámetro.

Entrada	• Catalogo. • Fecha inicial. • Fecha final.
Salidas	Un posible camino entre dos puntos turísticos dentro o fuera del territorio colombiano
Implementado (Sí/No)	Si se implementó y lo hizo Samuel

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Inicializar variables	O(1)
Recorrer aeropuertos	O(V)
Realizar comparaciones	O(1)
Algoritmo Dijkstra	O(V + E)

Obtener camino	O(V + E)
Recorrer vuelos del camino	O(V)
Obtener valor de la tabla de hash	O(1)
Añadir a la lista	O(1)

Pruebas Realizadas

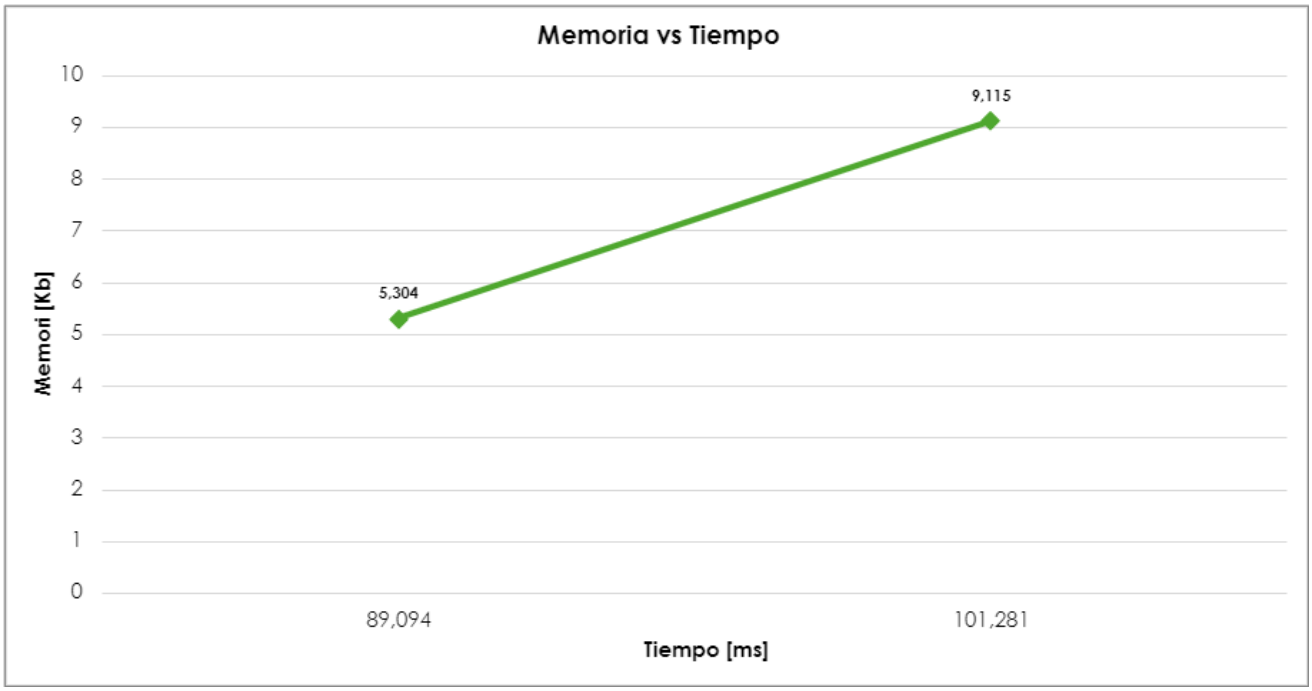
Las pruebas fueron realizadas tanto con un árbol RBT. El algoritmo de ordenamiento para la lista de ofertas fue Merge Sort. El tamaño de archivo fue large. La máquina utilizada cuenta con las siguientes capacidades.

Procesadores	Intel Core i7 -13700H
Memoria RAM (GB)	16 GB
Sistema operativo	Windows 11 Home

Tablas de datos

	Coordenadas	Aeropuertos	Distancia [Km]	Tiempo [Min]	Memoria [Kb]	Tiempo [ms]
FROM	4.6019927713895	2	678,71	655,853	5,304	89,094
	-74.06610470441926					
TO	10.507688799813222					
	-75.4706488665794					
FROM	12.542823	4	1353,02	1329,411	9,115	101,281
	-81.719398					
TO	3.104289					
	-75.220426					

Graficas



Análisis

El código funciona muy bien y tiene la complejidad adecuada para lo que se requiere. El hecho de usar Haversine como una función auxiliar es positivo para el desarrollo y consumo del requerimiento, por lo que se puede decir que es un requerimiento eficiente. El uso del algoritmo Dijkstra como preferencia también aporta beneficiosamente al requerimiento ya que impide que se recorran rutas innecesariamente largas que complican el algoritmo y aumentan el consumo de memoria.

Volver al índice

Requerimiento 2

Descripción

```python
def req_2(analyzer, origin, destination, map):
    """
    Función que soluciona el requerimiento 7
    """
    # Tabla de hash de aeropuertos
    airports = analyzer['airports']
    # Grafo de vuelos comerciales
    commercial = analyzer['commercial']
    # Lista de aeropuertos con mayor concurrecia comercial
    best_airports = me.getValue(get_entry(analyzer['best_airports'], om, 'commercial'))

    # Establecer aeropuertos de origen y destino
    origin_airport = None, 1000
    destination_airport = None, 100000

    for airport in lt.iterator(best_airports):
        distance = haversine(airport, origin)
        if distance < origin_airport[1]:
            origin_airport = airport, distance

        distance = haversine(airport, destination)
        if distance < destination_airport[1]:
            destination_airport = airport, distance

    if origin_airport[1] > 30 or destination_airport[1] > 30:
        return (origin_airport[1], destination_airport[1]), (origin_airport[0], destination_airport[0]), None

    searching_structure = bfs.BreathFirstSearch(commercial, origin_airport[0]['ICAO'])

    if not bfs.hasPathTo(searching_structure, destination_airport[0]['ICAO']):
        print("No hay ruta entre los aeropuertos")
        return None, (origin_airport[0], destination_airport[0]), None
    else:
        path = bfs.pathTo(searching_structure, destination_airport[0]['ICAO'])
```

Para cumplir con este requerimiento, debíamos hallar el camino con menos escalas (vértices visitados) entre dos destinos turísticos. Primero, usamos la formula Harvesine para hallar el aeropuerto mas cercano a las coordenadas ingresadas por el usuario. Sin embargo, si no hay ningún aeropuerto menor a 30 km de distancia no se ejecutara el código.

```
total_distance = origin_airport[1] + destination_airport[1]
total_airports = data_size(path, st)
flights_queue = qu.newQueue()

req_8_lst = lt.newList()
lt.addLast(req_8_lst, origin_airport[0])
lt.addLast(req_8_lst, destination_airport[0])

i = lt.size(path)

                                    (variable) i: Any | None
while i>1:
    vertexA = lt.getElement(path, i)
    vertexB = lt.getElement(path, i-1)


    airport_1 = me.getValue(get_entry(airports, mp, vertexA))
    airport_2 = me.getValue(get_entry(airports, mp, vertexB))

    total_distance += haversine(airport_1, airport_2)

    if airport_1['ICAO'] != origin_airport[0]['ICAO']:
        qu.enqueue(flights_queue, airport_1)
        lt.addLast(req_8_lst, airport_1)

    i -= 1

if map:
    req_8(req_8_lst)

return flights_queue, (origin_airport[0], destination_airport[0]), (total_distance, total_airports)
```

Después, tenemos que hallar el camino con menos escalas entre dos aeropuertos entre los dos aeropuertos encontrados usando el algoritmo de BFS. Finalmente, modelamos los datos para poder obtener con mayor precisión el origen, el destino y el peso.

Entrada	Rango de salarios
Salidas	Lista de trabajos filtrados y total de ofertas encontradas
Implementado (Sí/No)	Si, Andres

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Inicializar variables	O(1)
Recorrer aeropuertos	O(V)
Realizar comparaciones	O(1)
Algoritmo Dijkstra	O(V + E)
Obtener camino	O(V + E)
Recorrer vuelos del camino	O(V)
Obtener valor de la tabla de hash	O(1)
Añadir a la lista	O(1)

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

Pruebas Realizadas

La maquina utilizada cuenta con las siguientes capacidades.

	maquina
procesador	Ryzen 7 5700u
RAM (GB)	16 GB
Sistema operativo	Windows 10 home 64 bits

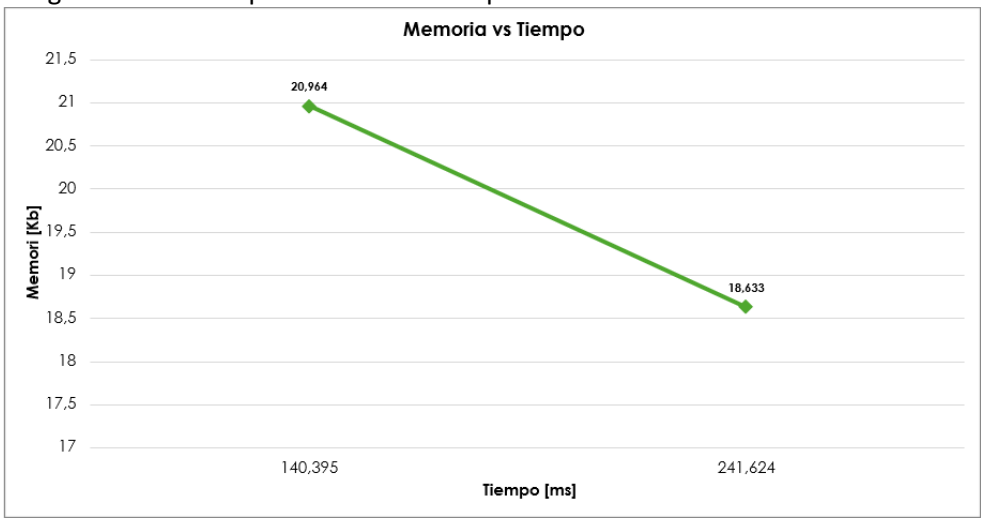
Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

	Coordenadas	Aeropuertos	Distancia [Km]	Tiempo [Min]	Memoria [Kb]	Tiempo [ms]
FROM	4.6019927713895	2	678,71	655,853	20,964	140,395
	-74.06610470441926					
TO	10.507688799813222					
	-75.4706488665794					
FROM	12.542823	3	1353,02	1329,411	18,633	241,624
	-81.719398					
TO	3.104289					
	-75.220426					

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

Este enfoque ha sido altamente efectivo en una variedad de contextos y ha demostrado su eficacia. Además, hemos desarrollado diferentes mensajes para situaciones específicas, como cuando no hay aeropuertos dentro de un radio de 30 km o cuando no existe una ruta directa entre los dos puntos seleccionados. El uso del algoritmo de BFS es especialmente apropiado en este caso, ya que encuentra la ruta con menos escalas, lo que garantiza una complejidad adecuada para manejar el requerimiento con cualquier conjunto de parámetros de entrada.

Volver al índice

Requerimiento 3

Descripción

En este requerimiento debíamos hallar la cobertura comercial desde el aeropuerto con mayor concurrencia en la menor distancia posible. Para esto hicimos uso del algoritmo Prim para hallar el subgrafo que recorre todos los vértices con el menor costo posible. Luego, con el algoritmo de Dijkstra hallamos el camino con menor costo para cada uno de los vértices de llegada que obtuvimos anteriormente. Finalmente, modelamos los datos para poder obtener con mayor precisión el origen, el destino y el peso.

```
def req_3(analyzer, map):
    """
    # Tabla de hash de aeropuertos
    airports = analyzer['airports']
    # Lista de aeropuertos con mayor concurrecia comercial
    best_airports = analyzer['best_airports']
    # Grafo de vuelos comerciales
    commercial = analyzer['commercial']

    important_arirport = lt.firstElement(me.getValue(get_entry(best_airports, mp, 'commercial')))
    commercial_coverage = djk.Dijkstra(commercial, important_arirport['ICAO'])

    flights = lt.newList()
    total_distance = 0

    # Ejecucion del requerimiento 8
    req_8_lst = lt.newList()
    lt.addLast(req_8_lst, important_arirport)

    for airport in lt.iterator(mp.keySet(prim.PrimMST(commercial, important_arirport['ICAO'])['edgeTo'])):

        if djk.hasPathTo(commercial_coverage, airport):
            # Ejecucion del requerimiento 8
            lt.addLast(req_8_lst, me.getValue(get_entry(airports, mp, airport)))

            # Hallar camino
            path = djk.pathTo(commercial_coverage, airport)
            distance = djk.distTo(commercial_coverage, airport)

            flights_queue = qu.newQueue()

            while not st.isEmpty(path):
                flight = st.pop(path)
                # Sumar la distancia del trayecto a la suma total de distancias
                total_distance += distance
```

```
            # Crear estructura para modelar los datos
            flight_structure = {
                'ORIGEN': me.getValue(get_entry(airports, mp, flight['vertexA'])),
                'DESTINO': me.getValue(get_entry(airports, mp, flight['vertexB'])),
                'DISTANCIA': distance
            }
            qu.enqueue(flights_queue, flight_structure)

        lt.addLast(flights, flights_queue)

total_flights = data_size(flights, lt)

# Ejecucion del requerimiento 8
if map:
    req_8(req_8_lst)

return flights, important_arirport, (total_flights, total_distance)
```

Entrada	Estructura datos cargada.
Salidas	Los vuelos de cobertura comercial, el aeropuerto con mayor concurrencia, el número de vuelos y la distancia total.
Implementado (Sí/No)	Si fue implementado por Juan David.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Inicializar variables	O(1)
Algoritmo Dijkstra	O(V + E)
Recorrer aeropuertos	O(V)
Realizar comparaciones	O(1)
Obtener camino	O(V + E)
Recorrer vuelos del camino	O(V)
Obtener valor de la tabla de hash	O(1)
Añadir a la lista	O(1)
TOTAL	***O(V + E)***

Pruebas Realizadas

Las pruebas fueron realizadas midiendo memoria. El algoritmo de ordenamiento para la lista de ofertas fue Merge Sort. La máquina utilizada cuenta con las siguientes capacidades.

Procesadores	11th Gen Intel® Core™ i5
Memoria RAM (GB)	8 GB
Sistema operativo	Windows 11 Home

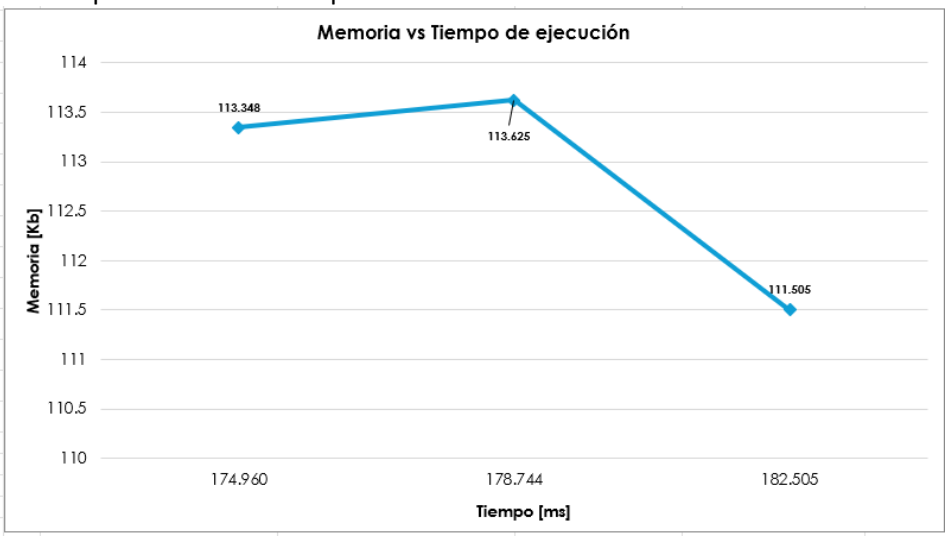
Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Vuelos	Distancia [Km]	Memoria [Kb]	Tiempo [ms]
64	5253.00	113.348	174.960
64	5253.00	113.625	178.744
64	5253.00	111.505	182.505

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

Este requerimiento funciona de manera muy eficiente, demostrando la gran rapidez de ejecución del algoritmo de Dijkstra. Gracias a esta eficiencia, podemos determinar todos los caminos con el costo mínimo desde el aeropuerto de mayor concurrencia de forma óptima. Esto permite encontrar rutas con costos mínimos con la mayor eficacia posible, optimizando así la gestión de rutas y mejorando la experiencia operativa del reto.

Volver al índice

Requerimiento 4

En este proyecto, nuestro objetivo era determinar la cobertura de carga desde el aeropuerto más concurrido, priorizando la distancia más corta posible. Para lograr esto, implementamos el algoritmo de Prim para identificar un subgrafo que conectara todos los aeropuertos con el menor costo. Posteriormente, empleamos el algoritmo de Dijkstra para calcular la ruta más económica desde cada aeropuerto hasta los destinos obtenidos anteriormente. Finalmente, refinamos nuestros datos para obtener información más precisa sobre los aeropuertos de origen y destino, así como sus respectivos costos asociados

Descripción

```python
def req_4(analyzer, map):
    """
    Función que soluciona el requerimiento 4
    """
    airports = analyzer["airports"]
    best_airports = analyzer["best_airports"]
    carga = analyzer["merchandise"]

    important_arirport = lt.firstElement(me.getValue(get_entry(best_airports, mp, 'merchandise')))
    merchandise_coverage = djk.Dijkstra(carga, important_arirport['ICAO'])

    flights = lt.newList()
    total_distance = 0

    # Ejecucion del requerimiento 8
    req_8_lst = lt.newList()
    lt.addLast(req_8_lst, important_arirport)

    for airport in lt.iterator(mp.keySet(prim.PrimMST(carga, important_arirport['ICAO'])['edgeTo'])):

        if djk.hasPathTo(merchandise_coverage, airport):
            # Ejecucion del requerimiento 8
            lt.addLast(req_8_lst, me.getValue(get_entry(airports, mp, airport)))

            # Hallar camino
            path = djk.pathTo(merchandise_coverage, airport)
            distance = djk.distTo(merchandise_coverage, airport)

            flights_queue = qu.newQueue()

            while not st.isEmpty(path):
                flight = st.pop(path)
                # Sumar la distancia del trayecto a la suma total de distancias
                total_distance += distance
```

```python
                # Crear estructura para modelar los datos
                flight_structure = {
                    'ORIGEN': me.getValue(get_entry(airports, mp, flight['vertexA'])),
                    'DESTINO': me.getValue(get_entry(airports, mp, flight['vertexB'])),
                    'DISTANCIA': distance
                }
                qu.enqueue(flights_queue, flight_structure)

            lt.addLast(flights, flights_queue)

    total_flights = data_size(flights, lt)

    # Ejecucion del requerimiento 8
    if map:
        req_8(req_8_lst)

    return flights, important_arirport, total_flights, total_distance
```

Entrada	Estructura de datos
Salidas	El aeropuerto más importante según la concurrencia de carga, distancia total, numero de trayectos posibles, la secuencia de trayectos con sus respectivos datos.
Implementado (Sí/No)	Si fue implementado, Andres

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Inicializar variables	O(1)
Algoritmo Dijkstra	O(V + E)
Recorrer aeropuertos	O(V)
Realizar comparaciones	O(1)
Obtener camino	O(V + E)
Recorrer vuelos del camino	O(V)
Obtener valor de la tabla de hash	O(1)
Añadir a la lista	O(1)
TOTAL	***O(V + E)***

Pruebas Realizadas

Las pruebas fueron realizadas midiendo memoria. El algoritmo de ordenamiento para la lista de ofertas fue Merge Sort. La máquina utilizada cuenta con las siguientes capacidades.

	maquina
procesador	Ryzen 7 5700u
RAM (GB)	16 GB
Sistema operativo	Windows 10 home 64 bits

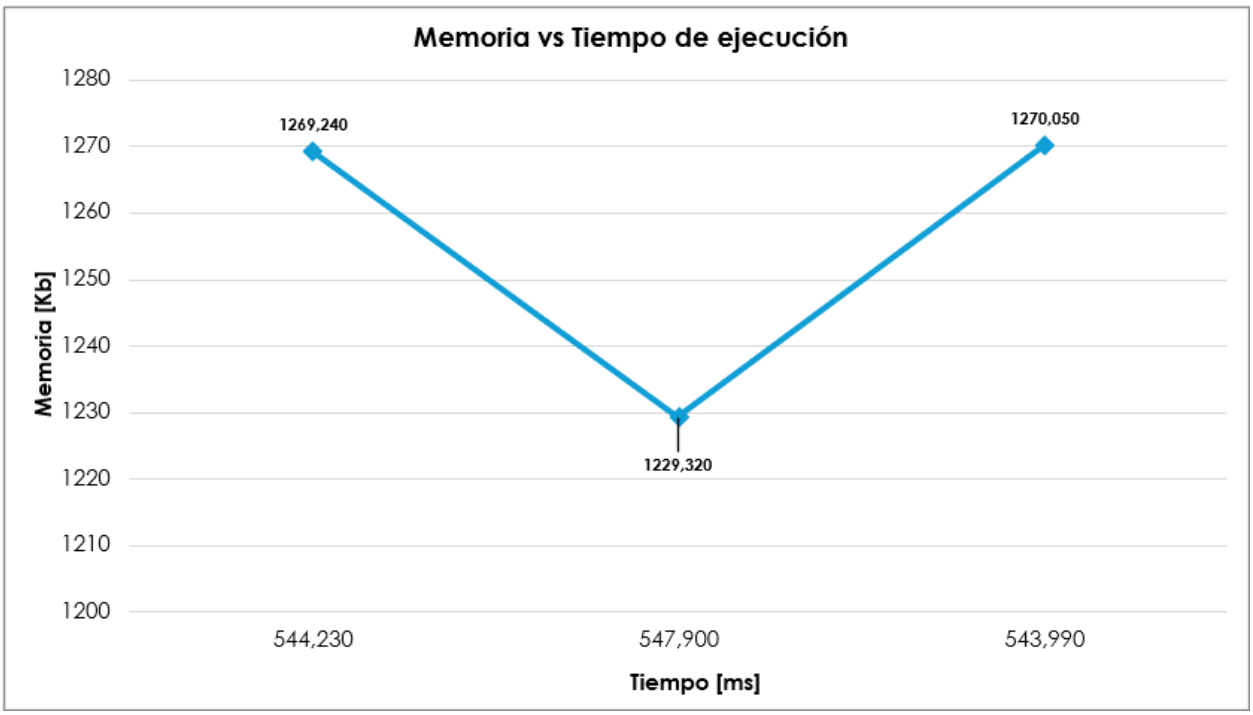
Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Vuelos	Distancia [Km]	Memoria [Kb]	Tiempo [ms]
70	123060,00	1269,240	544,230
70	123060,00	1229,320	547,900
70	123060,00	1270,050	543,990

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

Este requisito demuestra la eficacia del algoritmo de Dijkstra al funcionar de manera altamente eficiente, lo que nos permite identificar de manera óptima todos los caminos con el menor costo desde el aeropuerto más concurrido. Esta eficiencia nos habilita para encontrar rutas con costos mínimos de manera rápida y precisa, lo que mejora significativamente la gestión de rutas y optimiza la experiencia operativa del desafío

Volver al índice

Requerimiento 5

Descripción

```
def req_5(analyzer, map):
    """
    Función que soluciona el requerimiento 5
    """
        # Tabla de hash de aeropuertos
    airports = analyzer['airports']
    # Lista de aeropuertos con mayor concurrecia comercial
    best_airports = analyzer['best_airports']
    # Grafo de vuelos comerciales
    military = analyzer['military']

    important_arirport = lt.firstElement(me.getValue(get_entry(best_airports, mp, 'military')))
    military_coverage = djk.Dijkstra(military, important_arirport['ICAO'])

    flights = lt.newList()
    total_distance = 0

    # Ejecucion del requerimiento 8
    req_8_lst = lt.newList()
    lt.addLast(req_8_lst, important_arirport)
for airport in lt.iterator(mp.keySet(prim.PrimMST(military, important_arirport['ICAO'])['edgeTo'])):

    if djk.hasPathTo(military_coverage, airport):
        # Ejecucion del requerimiento 8
        lt.addLast(req_8_lst, me.getValue(get_entry(airports, mp, airport)))

        # Hallar camino
        path = djk.pathTo(military_coverage, airport)
        distance = djk.distTo(military_coverage, airport)

        flights_queue = qu.newQueue()

        while not st.isEmpty(path):
            flight = st.pop(path)
            # Sumar la distancia del trayecto a la suma total de distancias
            total_distance += distance

            # Crear estructura para modelar los datos
            flight_structure = {
                'ORIGEN': me.getValue(get_entry(airports, mp, flight['vertexA'])),
                'DESTINO': me.getValue(get_entry(airports, mp, flight['vertexB'])),
                'DISTANCIA': distance

            }
            qu.enqueue(flights_queue, flight_structure)

        lt.addLast(flights, flights_queue)

total_flights = data_size(flights, lt)
```

```
# Ejecucion del requerimiento 8
if map:
    req_8(req_8_lst)

return flights, important_arirport, (total_flights, total_distance)
```

Básicamente este algoritmo quería identificar las rutas que recorren los aviones desde el aeropuerto con mayor importancia militar en el menor tiempo posible. Para esto se uso el algoritmo Dijkstra para garantizar el menor tiempo y distancia. Primero se identificó el aeropuerto de mayor importancia militar al tratarse del que más tráfico de vuelos tenía y posteriormente se recorrió el grafo hasta todos los aeropuertos

Entrada	Estructura de datos
Salidas	Una red de trayectos para cubrir los aeropuertos de Colombia con el menor tiempo posible partiendo desde el aeropuerto con mayor importancia militar.
Implementado (Sí/No)	Si se implementó y lo hizo Samuel

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Inicializar variables	O(1)
Algoritmo Dijkstra	O(V + E)
Recorrer aeropuertos	O(V)
Realizar comparaciones	O(1)
Obtener camino	O(V + E)
Recorrer vuelos del camino	O(V)
Obtener valor de la tabla de hash	O(1)
Añadir a la lista	O(1)
TOTAL	***O(V + E)***

Pruebas Realizadas

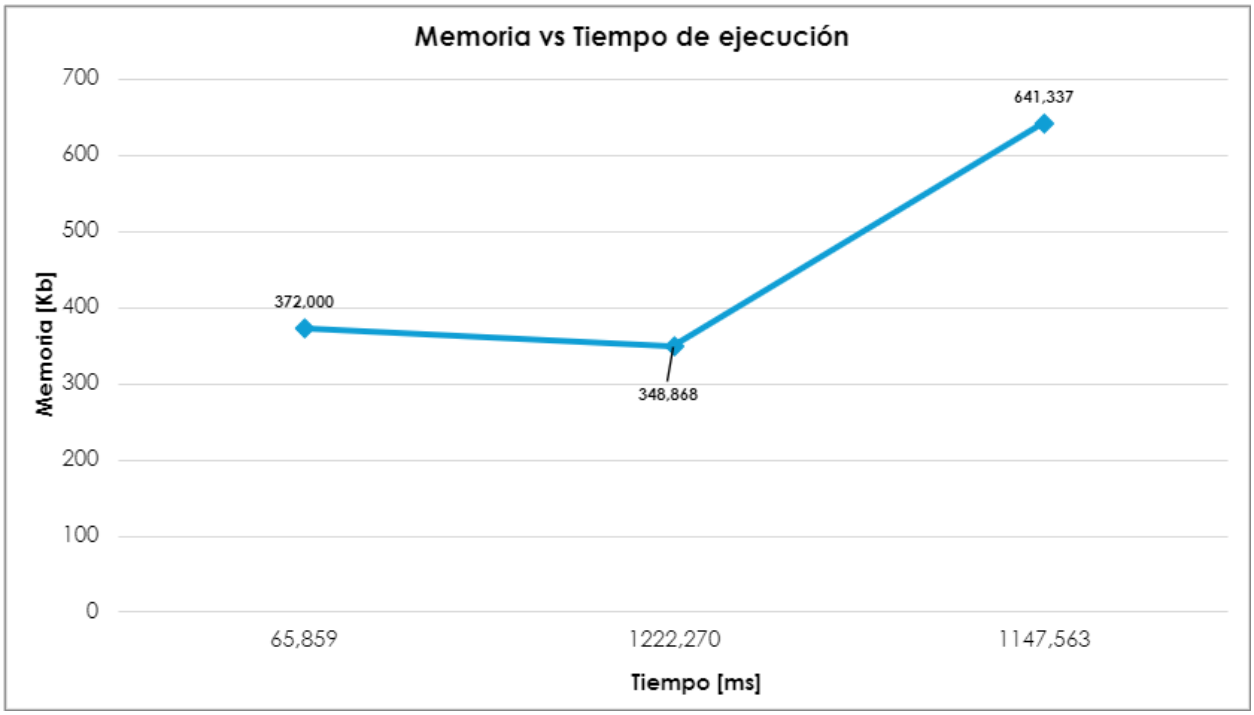
Las pruebas fueron realizadas tanto con un árbol RBT. El algoritmo de ordenamiento para la lista de ofertas fue Merge Sort. El tamaño de archivo fue large. La máquina utilizada cuenta con las siguientes capacidades.

Procesadores	Intel Core i7 -13700H
Memoria RAM (GB)	16 GB
Sistema operativo	Windows 11 Home

Tablas de datos

Vuelos	Distancia [Km]	Memoria [Kb]	Tiempo [ms]
69	2135.00	372,000	65,859
69	2135.01	348,868	1222,270
69	2135.02	641,337	1147,563

Graficas



Análisis

Considerando las pruebas realizadas y el resultado de la implementación podríamos decir que este código se implementó con complejidad adecuada y de manera muy efectiva, de modo que no involucra tiempos de respuesta ni mucho espacio en memoria.

Volver al índice

Requerimiento 6

Descripción

Para abordar este requerimiento, debemos identificar las rutas necesarias para cubrir los M aeropuertos más importantes de Colombia desde el aeropuerto con mayor concurrencia comercial. Para esto, obtuvimos la lista de los M aeropuertos colombianos organizados, de mayor a menor, por su

concurrencia comercial. Por ende, el primer aeropuerto que encontramos en la lista será el aeropuerto de partida.

```python
def req_6(analyzer, num_airports, map):
    """
    Función que soluciona el requerimiento 6
    """
    # Tabla de hash de aeropuertos
    airports = analyzer['airports']
    # Lista de aeropuertos con mayor concurrecia comercial
    best_airports = me.getValue(get_entry(analyzer['best_airports'], om, 'commercial'))

    colombian_airports = lt.newList()
    answer = lt.newList()

    for airport in lt.iterator(best_airports):
        # Encontrar los aeropuertos colombianos
        if airport['PAIS'].upper() == 'COLOMBIA':
            lt.addLast(colombian_airports, airport)

    # Extraer el aeropuerto con mayor concurrencia
    important_airport = lt.removeFirst(colombian_airports)
    # Verificar que la lista sea menor o igual al tamaño recibido por parametro
    if data_size(colombian_airports, lt) > num_airports:
        colombian_airports = get_sublist(colombian_airports, 1, num_airports)

    # Calcular los caminos de costo mínimo
    military_coverage = djk.Dijkstra(analyzer['military'], important_airport['ICAO'])
```

Luego, usamos el algoritmo de Dijkstra para hallar el camino con la menor distancia posible para el resto de los aeropuertos que tenemos en la lista. Finalmente, modelamos los datos para poder obtener con mayor precisión el origen, el destino y el peso.

```python
    # Recorrer los n aeropuertos que se desean cubrir
    for airport in lt.iterator(colombian_airports):
        # Hallar el camino hasta el aeropuerto
        path = djk.pathTo(military_coverage, airport['ICAO'])

        airports_queue = qu.newQueue()
        flights_queue = qu.newQueue()

        if path is None:
            lt.addLast(answer, { 'DISTANCE': 0 })

        else:
            while not st.isEmpty(path):
                flight = st.pop(path)

                # Obtener los datos del aeropuerto ubicado en el vertice A
                airport_structure = me.getValue(get_entry(airports, mp, flight['vertexA']))
                qu.enqueue(airports_queue, airport_structure)

                # Crear estructura para modelar los datos
                flight_structure = {
                    'ORIGEN': flight['vertexA'],
                    'DESTINO': flight['vertexB']
                }
                qu.enqueue(flights_queue, flight_structure)

            # Crear estructura para modelar los datos
            airport_coverage = {
                'AIRPORTS': airports_queue,
                'FLIGHTS': flights_queue,
                'DISTANCE': djk.distTo(military_coverage, airport['ICAO'])
            }
            lt.addLast(answer, airport_coverage)

    # Volver a añadir el aeropuerto con mayor concurrencia a la lista
    lt.addFirst(colombian_airports, important_airport)

    # Ejecucion del requerimiento 8
    if map:
        req_8(colombian_airports)

    return answer, colombian_airports
```

Entrada	Estructura datos cargada, número de aeropuertos a cubrir.
Salidas	Las rutas para cubrir los M aeropuertos, los aeropuertos mas importantes de Colombia.

Implementado (Sí/No)	Si fue implementado por Juan David.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Inicializar variables	O(1)
Recorrer aeropuertos	O(V)
Realizar comparaciones	O(1)
Algoritmo Dijkstra	O(V + E)
Obtener camino	O(V + E)
Recorrer vuelos del camino	O(V)
Obtener valor de la tabla de hash	O(1)
Añadir a la lista	O(1)
TOTAL	***O(V + E)***

Pruebas Realizadas

Las pruebas fueron realizadas midiendo memoria. El algoritmo de ordenamiento para la lista de ofertas fue Merge Sort. La máquina utilizada cuenta con las siguientes capacidades.

Procesadores	11th Gen Intel® Core™ i5
Memoria RAM (GB)	8 GB
Sistema operativo	Windows 11 Home

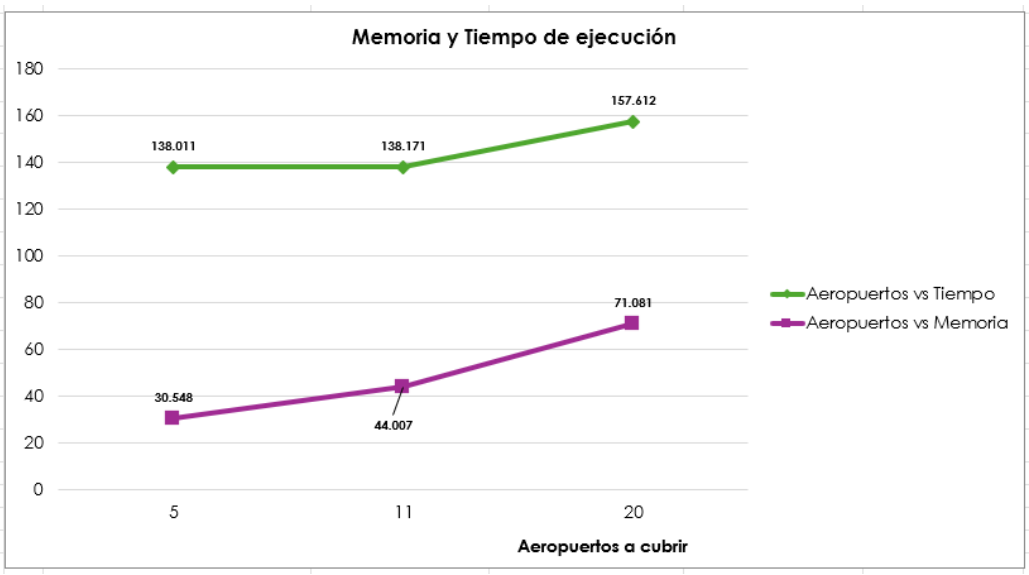
Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Aeropuertos a cubrir	Memoria [Kb]	Tiempo [ms]
5	30.548	138.011
11	44.007	138.171
20	71.081	157.612

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

En este requerimiento, podemos observar que a medida que aumenta la cantidad de aeropuertos que se desean cubrir, también aumentan la memoria ocupada y el tiempo de ejecución. Sin embargo, el uso del algoritmo de Dijkstra optimiza significativamente este proceso, lo cual se traduce en una ejecución muy rápida. Dijkstra permite encontrar los caminos con el costo mínimo de manera eficiente, asegurando que incluso con un gran número de aeropuertos, el rendimiento del sistema se mantenga elevado.

Volver al índice

Requerimiento 7

Descripción

Para cumplir con este requerimiento, debíamos hallar el camino más corto en tiempo entre dos destinos turísticos. Primero, usamos la formula Harvesine para hallar el aeropuerto mas cercano a las coordenadas ingresadas por el usuario. Sin embargo, si no hay ningún aeropuerto menor a 30 km de distancia no se ejecutara el código.

```python
def req_7(analyzer, origin, destination, map):
    """
    Función que soluciona el requerimiento 7
    """
    # Tabla de hash de aeropuertos
    airports = analyzer['airports']
    # Grafo de vuelos comerciales
    commercial = analyzer['commercial']
    # Lista de aeropuertos con mayor concurrecia comercial
    best_airports = me.getValue(get_entry(analyzer['best_airports'], om, 'commercial'))

    # Establecer aeropuertos de origen y destino
    origin_airport = None, 100000
    destination_airport = None, 100000

    for airport in lt.iterator(best_airports):
        # Hallar el aeropuerto mas cercano a las coordenadas
        distance = haversine(airport, origin)
        if distance < origin_airport[1]:
            origin_airport = airport, distance

        distance = haversine(airport, destination)
        if distance < destination_airport[1]:
            destination_airport = airport, distance

    # Si no hay un aeropuerto menor a los 30 Km, no se ejecuta la busqueda
    if origin_airport[1] > 30 or destination_airport[1] > 30:
        return (origin_airport[1], destination_airport[1]), (origin_airport[0], destination_airport[0]), None
```

```python
searching_strucuture = djk.Dijkstra(commercial, origin_airport[0]['ICAO'])

# Verificar si hay un camino entre los dos aeropuertos
if not djk.hasPathTo(searching_strucuture, destination_airport[0]['ICAO']):
    return None, (origin_airport[0], destination_airport[0]), None
else:
    # Hallar el camino hasta el aeropuerto
    path = djk.pathTo(searching_strucuture, destination_airport[0]['ICAO'])

# Iniciar variables para guardar los datos que pide el requerimiento
total_distance = origin_airport[1] + destination_airport[1]
total_time = djk.distTo(searching_strucuture, destination_airport[0]['ICAO'])
total_airports = data_size(path, st) + 1
flights_queue = qu.newQueue()

# Ejecucion del requerimiento 8
req_8_lst = lt.newList()
lt.addLast(req_8_lst, origin_airport[0])
lt.addLast(req_8_lst, destination_airport[0])

while not st.isEmpty(path):
    flight = st.pop(path)

    airport_1 = me.getValue(get_entry(airports, mp, flight['vertexA']))
    airport_2 = me.getValue(get_entry(airports, mp, flight['vertexB']))
    # Sumar la distancia del trayecto a la suma total de distancias
    total_distance += haversine(airport_1, airport_2)

    # Agregar aeropuerto intermedio a la cola
    if airport_1['ICAO'] != origin_airport[0]['ICAO']:
        qu.enqueue(flights_queue, airport_1)
        lt.addLast(req_8_lst, airport_1)

# Ejecucion del requerimiento 8
if map:
    req_8(req_8_lst)

return flights_queue, (origin_airport[0], destination_airport[0]), (total_time, total_distance, total_airports)
```

Después, tenemos que hallar el camino de costo mínimo entre los dos aeropuertos entre los dos aeropuertos encontrados usando el algoritmo de Dijkstra. Finalmente, modelamos los datos para poder obtener con mayor precisión el origen, el destino y el peso.

Entrada	Estructura datos cargada, coordenadas de origen y de destino.
Salidas	La secuencia de vuelos, los aeropuertos de origen y de destino, el tiempo, la distancia recorrida y el número de aeropuertos visitados.
Implementado (Sí/No)	Si fue implementado por Juan David.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Inicializar variables	O(1)
Recorrer aeropuertos	O(V)
Realizar comparaciones	O(1)
Algoritmo Dijkstra	O(V + E)
Obtener camino	O(V + E)
Recorrer vuelos del camino	O(V)
Obtener valor de la tabla de hash	O(1)
Añadir a la lista	O(1)
TOTAL	***O(V + E)***

Pruebas Realizadas

Las pruebas fueron realizadas midiendo memoria. El algoritmo de ordenamiento para la lista de ofertas fue Merge Sort. La máquina utilizada cuenta con las siguientes capacidades.

Procesadores	11th Gen Intel® Core™ i5
Memoria RAM (GB)	8 GB
Sistema operativo	Windows 11 Home

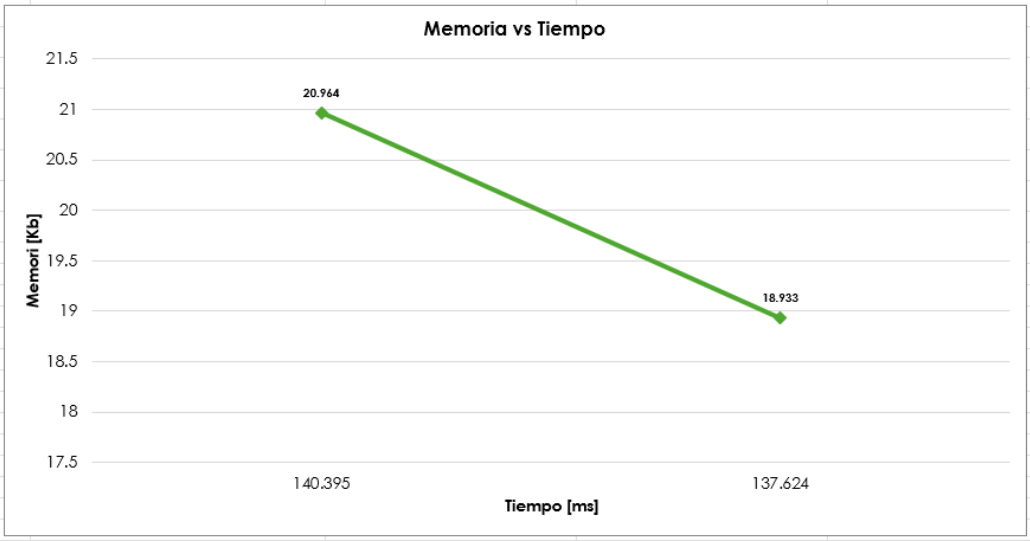
Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

	Coordenadas	Aeropuertos	Distancia [Km]	Tiempo [Min]	Memoria [Kb]	Tiempo [ms]
FROM	4.6019927713895	2	678.71	655.853	20.964	140.395
	-74.06610470441926					
TO	10.507688799813222					
	-75.4706488665794					
FROM	12.542823	4	1353.02	1329.411	18.933	137.624
	-81.719398					
TO	3.104289					
	-75.220426					

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

Este requerimiento funciona muy bien y ha demostrado ser eficaz en diversas situaciones. Además, hemos implementado diferentes mensajes para casos específicos: cuando no se encuentran aeropuertos a menos de 30 km o cuando no hay una ruta entre los dos puntos. El uso del algoritmo de Dijkstra es particularmente adecuado para este propósito, ya que encuentra la ruta de menor costo de manera eficiente. Esto resulta en una complejidad adecuada para ejecutar el requerimiento con cualquier conjunto de parámetros de entrada.

Volver al índice

Requerimiento 8

Descripción

Para llevar a cabo el requerimiento 8, empleamos la extensión Folium, la cual nos permite crear mapas interactivos y añadir marcadores. El código comienza creando un mapa utilizando la función Map().

```
# Crear mapa
map = folium.Map()

for airport in lt.iterator(airports):
    # Obtener las coordenadas como una tupla (latitud, longitud)
    coordenates = (airport["LATITUD"],airport["LONGITUD"])
    # Mensaje que se mostrara al hacer click sobre un marcador
    message = airport["NOMBRE"] + ' - ' + airport['ICAO']

    # Añadir marcador al mapa
    folium.Marker(location=coordenates,
                  tooltip=airport['CIUDAD'],
                  popup=message).add_to(map)

# Guardar mapa para poder visualizarlo
map.save("mapa_req_8.html")
```

Luego, iteramos sobre la lista de aeropuertos recibida como parámetro para obtener las coordenadas (latitud, longitud) de cada una. Posteriormente, añadimos un mensaje que se mostrará al hacer clic sobre un marcador, el cual incluye el nombre del aeropuerto y su código ICAO. Finalmente, guardamos

el mapa en formato HTML para poder visualizarlo posteriormente en nuestro navegador. Cada uno de nuestros requerimientos llama a la función req_8() para así poder crear el mapa interactivo.

Entrada	Lista de aeropuertos.
Salidas	Mapa en formato .html
Implementado (Sí/No)	Si fue implementado por Juan David, Samuel y Ándres.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Crear mapa	O(1)
Iterar sobre los aeropuertos	O(n)
Obtener coordenadas	O(1)
Inicializar mensaje	O(1)
Añadir marcador	O(1)
Guardar mapa	O(1)
TOTAL	***O(n)***

Análisis

Para el requerimiento 8, las pruebas realizadas son las mismas que para cada uno de los requerimientos anteriores, dado que esta función está implementada en todos ellos. Respecto a la complejidad de nuestro algoritmo, presenta una complejidad ideal de O(n), lo que garantiza su eficacia. Sin embargo, es importante destacar que al intentar agregar un gran número de aeropuertos al mapa interactivo, nos enfrentamos al problema de que el navegador no puede cargarlo. Esto puede deberse a la sobrecarga de datos y recursos que implica la visualización de un gran conjunto de marcadores en el mapa.

Volver al índice