

ANÁLISIS DEL RETO

Mateo Sánchez Zapata, 202321354, m.sanchezz@uniandes.edu.co

Santiago Garzon Garcia, 202512373, S.garzong2@uniandes.edu.co

Santiago Ismael Escobar Maidana, 202516956, si.escobar@uniandes.edu.co

Requerimiento <>n><

Plantilla para el documentar y analizar cada uno de los requerimientos.

Descripción

Breve descripción de como abordaron la implementación del requerimiento

Entrada	Parámetros necesarios para resolver el requerimiento.
Salidas	Respuesta esperada del algoritmo.
Implementado (Sí/No)	Si se implementó y quien lo hizo.

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	$O(...)$

Análisis

Análisis de resultados de la implementación, tener cuenta las pruebas realizadas y el análisis de complejidad.

Requerimiento Ejemplo

Descripción

```
def get_data(data_structs, id):
    """
    Retorna un dato a partir de su ID
    """
    pos_data = lt.isPresent(data_structs["data"], id)
    if pos_data > 0:
        data = lt.getElement(data_structs["data"], pos_data)
        return data
    return None
```

Este requerimiento se encarga de retornar un dato de una lista dado su ID. Lo primero que hace es verificar si el elemento existe. Dado el caso que exista, retorna su posición, lo busca en la lista y lo retorna. De lo contrario, retorna None.

Entrada	Estructuras de datos del modelo, ID.
Salidas	El elemento con el ID dado, si no existe se retorna None
Implementado (Sí/No)	Sí. Implementado por Juan Andrés Ariza

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	Complejidad
Buscar si el elemento existe (isPresent)	$O(n)$
Obtener el elemento (getElement)	$O(1)$
TOTAL	$O(n)$

Análisis

A pesar de que obtener un elemento en un *ArrayList*, dada su posición, tiene complejidad constante, la implementación de este requerimiento tiene un orden lineal $O(n)$. Esto debido a que, lo primero que se hace es verificar si el elemento hace parte de la lista. Específicamente, a la hora de buscar un elemento en una lista, en el peor de los casos es necesario recorrer toda la lista, es decir, complejidad lineal.

Este comportamiento se puede evidenciar experimentalmente en la gráfica. Ya que, gracias a que los datos no se encuentran tan dispersos con respecto a la línea de tendencia, la curva coincide con el comportamiento lineal esperado.

Carga de datos

Descripción

Esta función es la encargada de transformar los registros crudos del archivo CSV en las estructuras de datos del grafo. El proceso inicia leyendo todas las filas y ordenándolas cronológicamente mediante Merge Sort para garantizar una secuencia temporal correcta. Posteriormente, se itera linealmente sobre los registros para realizar una agrupación espacio-temporal: si un evento ocurre cerca (geoespacial y temporalmente) del anterior, se agrupa en el mismo nodo; de lo contrario, se crea un nuevo vértice. Simultáneamente, se rastrea el identificador de cada grulla mediante mapas hash para detectar transiciones entre nodos y construir arcos de movimiento y conexiones hídricas. Finalmente, se consolidan los arcos calculando promedios de pesos y se genera un reporte estadístico.

Entrada	Nombre del archivo CSV.
Salidas	Catálogo con los dos grafos poblados ("mov_migratorios" y "recursos_hidricos") y un diccionario con el reporte de carga (estadísticas y muestras de datos)
Implementado (Sí/No)	Sí por todos los estudiantes

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	Complejidad
Paso 1: Lectura del archivo CSV y formateo de fechas	$O(N)$
Paso 2: Ordenamiento de registros por fecha (Merge Sort)	$O(N \log N)$
Paso 3: Agrupación de nodos y mapeo de arcos (Bucle principal)	$O(N)$
Paso 4: Construcción final de arcos en el grafo (Iteración sobre mapas auxiliares)	$O(V + E)$
Paso 5: Generación de reporte (Muestreo constante)	$O(1)$
TOTAL	<i>TOTAL TEMPORAL $O(N \log N)$</i> <i>TOTAL ESPACIAL $O(N + V + E)$</i>

Análisis

El algoritmo de carga está dominado temporalmente por el proceso de ordenamiento, con una complejidad de $O(N \log N)$, donde N es el número total de registros en el archivo CSV. Esto es necesario para procesar la secuencia temporal de los eventos. El agrupamiento posterior se realiza en tiempo lineal $O(N)$ gracias al uso de mapas hash (tablas de dispersión) para gestionar el estado de las grullas y los acumuladores de arcos en tiempo constante $O(1)$ amortizado. La construcción final de los grafos depende de la cantidad de vértices (V) y aristas (E) resultantes, que siempre será menor o igual a N .

En cuanto a la complejidad espacial, el orden es $O(N + V + E)$. Se requiere $O(N)$ para almacenar la lista inicial de filas crudas en memoria antes de procesarlas. Adicionalmente, se utiliza espacio $O(V + E)$ para mantener las estructuras del grafo (listas de adyacencia) y los mapas auxiliares necesarios para calcular los promedios de los pesos de los arcos sin necesidad de recorridos cuadráticos.

Diagrama

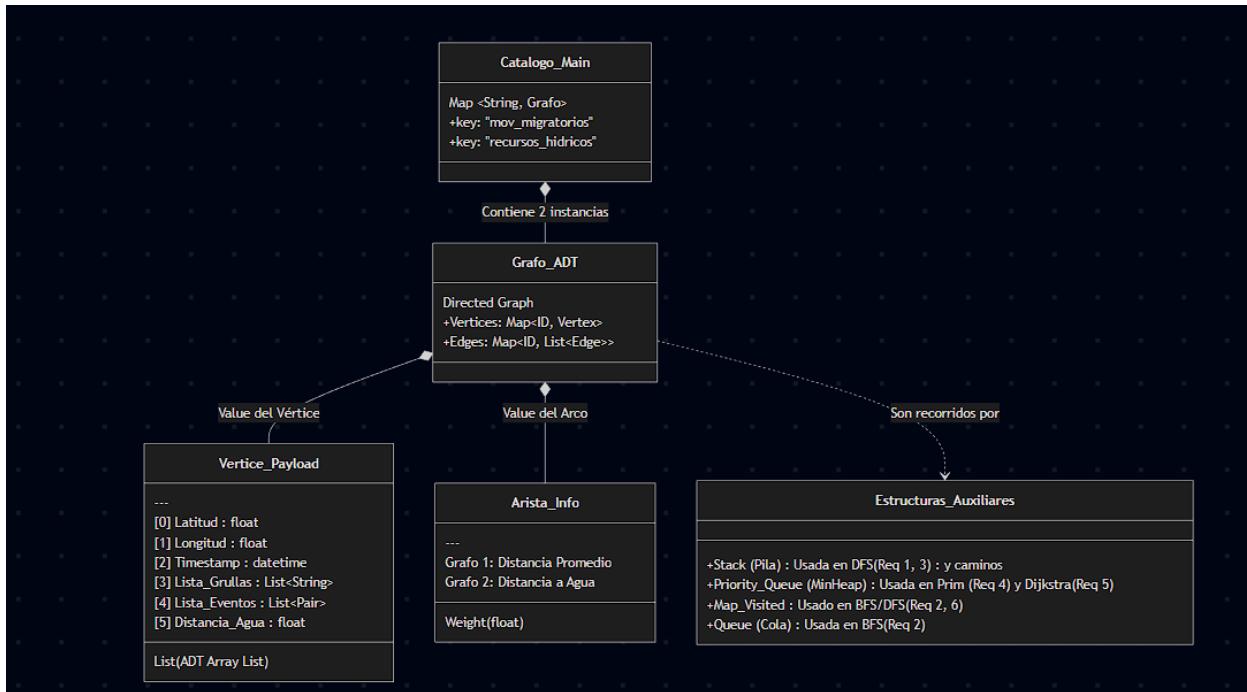


Diagrama de Estructuras de Datos: El diagrama presenta la arquitectura de datos persistente en memoria tras la ejecución de `load_data`.

Catalogo: Se utiliza una Tabla de Símbolos (Map Linear Probing) que actúa como contenedor principal, almacenando los dos grafos dirigidos requeridos: uno para movimientos migratorios y otro para recursos hídricos.

Nivel de Grafo: Ambos grafos comparten la misma estructura de vértices. Se implementan mediante listas de adyacencia.

Nivel de Vértice: La información asociada a cada nodo no es un valor simple, sino una Lista ADT (ArrayList) estructurada con índices fijos (0-5) que almacena: coordenadas geográficas, marca temporal, una sub-lista con los IDs de las grullas presentes, una sub-lista con el historial de eventos agrupados y el promedio de distancia al agua.

Estructuras para Requerimientos: Para la resolución de los algoritmos (Req 1 al 6) se usan estructuras dinámicas temporales sobre el grafo, tales como Pilas (Stack) para reconstrucción de caminos y ordenamientos topológicos, Colas de Prioridad (MinHeap) para los algoritmos Prim y Dijkstra, y Mapas auxiliares para el control de visitados en recorridos BFS/DFS.

Requerimiento <<1>>

Descripción

Este requerimiento establece una ruta de conectividad simple para un individuo específico entre dos coordenadas geográficas. La implementación comienza localizando los nodos del grafo más cercanos al origen y destino mediante búsqueda lineal. Posteriormente, ejecuta el algoritmo de búsqueda en profundidad (DFS) desde el nodo de origen para explorar la conectividad. Si existe un camino, se reconstruye la ruta y se itera sobre la lista de nodos resultante para calcular la distancia total acumulada, identificar el primer avistamiento de la grulla solicitada y extraer información detallada (latitud, longitud, eventos) de cada vértice involucrado.

Entrada	Catálogo con el grafo de movimientos migratorios, Latitud y Longitud de origen, Latitud y Longitud de destino, Identificador de la grulla (crane_id).
Salidas	Diccionario con validación de errores, nodos de origen/destino, primer nodo de avistamiento, longitud del camino (pasos), distancia total (peso), y subconjuntos detallados de los primeros y últimos nodos de la ruta.
Implementado (Sí/No)	Si se implementó por Mateo Sánchez.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	Complejidad
Paso 1: Encontrar nodos más cercanos (Búsqueda lineal)	$O(V)$
Paso 2: Ejecución del algoritmo DFS	$O(V + E)$
Paso 3: Verificación de conectividad y reconstrucción del camino	$O(V)$
Paso 4: Recorrido del camino reconstruido	$O(L)$
Paso 5: Extracción y conversión de grullas por nodo (dentro del bucle del camino)	$O(L \times G)$
TOTAL	TOTAL TEMPORAL $O(V + E)$ TOTAL ESPACIAL $O(V)$

Análisis

El requerimiento se implementó correctamente utilizando el algoritmo DFS para determinar la existencia de una ruta entre dos puntos. La complejidad temporal dominante es $O(V + E)$ debido a la naturaleza de la exploración del grafo mediante DFS. La búsqueda de los nodos iniciales añade un costo lineal $O(V)$, que no altera el orden asintótico general. El procesamiento posterior depende de la longitud del camino encontrado (L) y la cantidad promedio de grullas por nodo (G), siendo eficiente para caminos que no cubran la totalidad del grafo. Espacialmente, el algoritmo mantiene un orden $O(V)$ para almacenar el mapa de visitados ([visited](#)) y la estructura del camino reconstruido.

Requerimiento <>2>>

Descripción

Este requerimiento detecta los movimientos de un nicho biológico delimitados por un área de interés. La implementación inicia localizando los nodos del grafo más próximos a las coordenadas geográficas de origen y destino mediante una búsqueda lineal. Posteriormente, ejecuta el algoritmo BFS para establecer la conectividad y reconstruir la ruta entre los puntos. Una vez obtenido el camino, se recorre la pila de nodos resultante para calcular la distancia acumulada, verificar la pertenencia al radio de interés mediante la fórmula Haversine y recopilar estadísticas detalladas como grullas y eventos por vértice.

Entrada	Catálogo con el grafo, Punto migratorio de origen, Punto migratorio de destino, Radio en km
Salidas	Diccionario con el último nodo dentro del radio, distancia total, total de puntos, y lista detallada de vértices con sus grullas y distancias.
Implementado (Sí/No)	Sí, por Santiago Escobar.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	Complejidad
Paso 1: Encontrar nodos más cercanos	$O(V)$
Paso 2: Ejecución de BFS	$O(V + E)$
Paso 3: Reconstrucción del camino	$O(V)$
Paso 4: Procesamiento del camino y verificación de radio	$O(L)$
Paso 5: Extracción de datos de grullas por nodo	$O(G)$
TOTAL	TOTAL TEMPORAL $O(V + E)$
	TOTAL ESPACIAL $O(V)$

Análisis

El requerimiento se implementó correctamente utilizando el algoritmo BFS. La complejidad temporal dominante es $O(V + E)$ debido a la exploración del grafo. En cuanto a la complejidad espacial, el

algoritmo es eficiente con un orden $O(V)$, ya que requiere almacenar estructuras auxiliares lineales para el mapa de visitados, la cola de procesamiento y la pila para la reconstrucción del camino. El procesamiento posterior de verificación de radio opera sobre el subconjunto de nodos del camino (L), sin incrementar el orden de complejidad general.

Requerimiento <<3>>

Descripción

Este requerimiento identifica **posibles rutas migratorias** desde un punto de origen específico dentro del grafo de movimientos migratorios.

La implementación verifica primero si el punto existe, determina si el grafo contiene ciclos, realiza ordenamiento topológico cuando es posible y usa **BFS** para encontrar rutas desde el origen hacia sumideros o nodos alcanzables.

Finalmente, construye la información detallada de cada ruta (distancias, grullas, eventos, etc.).

Entrada	Catálogo con el grafo , Punto migratorio desde el cual inicial las rutas
Salidas	Diccionario con información sobre rutas migratorias, ciclos, orden topológico, detalles de puntos y distancias, tiempos de ejecución.
Implementado (Sí/No)	Si se implementó , por Santiago Garzon

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	Complejidad
Paso 1	Verificar existencia del vértice $O(1)$
Paso 2	Comprobación de ciclos $O(V + E)$
Paso 3	Ordenamiento topológico $O(V + E)$
Paso 4	BFS desde el punto de origen $O(V + E)$
Paso 5	Recorrer todos los vértices para buscar sumideros $O(V)$
Paso 6	Construir rutas (para cada sumidero) $O(L)$
Paso 7	Obtener información de cada vértice dentro de cada ruta $O(L)$
Paso 8	Calcular distancias entre nodos (acceso por mapa) $O(1)$
Paso 9	Ordenar rutas por longitud $O(R \log R)$
TOTAL	$O(V + E + R \log R)$

Análisis

El requerimiento se implementó correctamente y permite identificar rutas migratorias desde un punto dado, verificando primero la existencia del nodo, comprobando si el grafo contiene ciclos y realizando un ordenamiento topológico cuando es posible. Mediante BFS se determinan todos los nodos alcanzables desde el origen, se identifican sumideros o, en su defecto, los puntos más lejanos, y se construyen rutas completas incluyendo información detallada de cada nodo, como coordenadas, grullas, eventos y distancias acumuladas. Las pruebas evidenciaron que el algoritmo funciona adecuadamente en diferentes escenarios, y que su desempeño es eficiente gracias al uso de BFS, estructuras hash y recorridos lineales sobre los nodos alcanzables, logrando así una solución consistente y escalable.

Requerimiento <<4>>

Descripción

Este requerimiento construye un corredor hídrico óptimo calculando el Árbol de Expansión Mínima (MST) a partir de un punto de interés. La implementación localiza el nodo hídrico más cercano y ejecuta el algoritmo de Prim para conectar los vértices del grafo minimizando los pesos de las aristas (distancias). Una vez generado el orden de visita y la estructura del MST, se recorre la totalidad de los nodos resultantes para consolidar estadísticas: cálculo de la distancia total del corredor, conteo de individuos únicos (grullas) utilizando un conjunto para evitar duplicados, y estructuración de los datos geográficos de cada punto hídrico.

Entrada	Catálogo con el grafo de recursos hídricos, Latitud de origen, Longitud de origen.
Salidas	Diccionario con estadísticas del corredor (puntos totales, individuos únicos, distancia acumulada), detalles de los primeros y últimos nodos, y tiempo de ejecución.
Implementado (Sí/No)	Sí se implementó por todos

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	Complejidad
Paso 1: Verificación de grafo vacío	$O(1)$
Paso 2: Encontrar nodo hídrico más cercano (Búsqueda lineal)	$O(V)$
Paso 3: Ejecución del algoritmo de Prim (Asumiendo uso de cola de prioridad)	$O(E \log V)$
Paso 4: Procesamiento de resultados del MST (Recorrido de vértices visitados)	$O(V)$
Paso 5: Conversión de listas de grullas y uso de Set para conteo únicos	$O(V \times G)$
TOTAL	<i>TOTAL TEMPORAL O(E log V)</i>

	TOTAL ESPACIAL O(V)
--	----------------------------

Análisis

El requerimiento implementa correctamente la optimización de redes utilizando el algoritmo de Prim. La complejidad temporal está dominada por la construcción del MST, que es $O(E \log V)$ asumiendo una implementación eficiente con colas de prioridad (o $O(V^2)$ si se usara una implementación densa sin estructuras auxiliares óptimas, aunque el estándar es $E \log V$). La fase de post-procesamiento recorre todos los nodos (V) del árbol generado para extraer métricas, realizando operaciones dependientes de la cantidad de grullas (G) en cada nodo. El uso de un conjunto para contar individuos únicos asegura la corrección de los datos sin incrementar el orden de complejidad más allá del procesamiento lineal de los datos de cada vértice. Espacialmente, requiere $O(V)$ para almacenar las estructuras del MST.

Requerimiento <<5>>

Descripción

Este requerimiento identifica la ruta migratoria más eficiente entre dos puntos basándose en un criterio de optimización (distancia o recursos hídricos). La implementación selecciona dinámicamente el grafo ponderado correspondiente y localiza los vértices de origen y destino mediante aproximación geográfica. El núcleo del procesamiento utiliza el algoritmo de Dijkstra para calcular los costos mínimos. Tras validar la existencia de una ruta, se reconstruye el camino desde la estructura de resultados y se procesa secuencialmente para acumular el costo total, contar segmentos y extraer la información de grullas asociada a cada punto migratorio.

Entrada	Catálogo con los grafos, Punto migratorio de origen, Punto migratorio de destino, Criterio (distancia/agua)
Salidas	Diccionario con costo total, total de puntos, total de segmentos y detalle de los vértices de la ruta óptima.
Implementado (Sí/No)	Sí se implementó por todos los integrantes.

Análisis de complejidad

Analisis de complejidad de cada uno de los pasos del algoritmo

Pasos	Complejidad
Paso 1: Selección del grafo	$O(1)$
Paso 2: Encontrar nodos más cercanos	$O(V)$
Paso 3: Ejecución de Dijkstra	$O(E \log V)$
Paso 4: Recuperación de costo y camino	$O(V)$
Paso 5: Procesamiento de detalles de la ruta	$O(L)$
Paso 6: Copia de listas de grullas	$O(G)$
TOTAL	TOTAL TEMPORAL O($E \log V$)

TOTAL ESPACIAL $O(V)$

Análisis

La eficiencia temporal de este requerimiento es $O(E \log V)$ gracias al uso de una cola de prioridad (Min-Heap) en el algoritmo de Dijkstra. Respecto a la complejidad espacial, el requerimiento mantiene un orden $O(V)$, necesario para almacenar las estructuras de datos auxiliares: el arreglo de distancias acumuladas, el mapa de predecesores para reconstruir la ruta y la propia cola de prioridad indexada. Estas estructuras aseguran que el consumo de memoria crezca linealmente con el número de vértices, haciendo la solución escalable para grafos grandes.

Requerimiento <<6>>

Descripción

Este requerimiento identifica subredes hídricas o componentes conectados dentro del grafo de recursos hídricos del nicho biológico. La implementación verifica primero que el punto de origen exista y que el grafo no esté vacío; luego recorre todos los nodos del grafo usando BFS para agruparlos en componentes independientes, marcando los vértices visitados para evitar exploraciones repetidas. Para cada subred encontrada se recopila información detallada: límites geográficos, nodos pertenecientes, grullas registradas, eventos, posiciones, distancias hídricas y orden temporal. Finalmente, las subredes se organizan por tamaño y se incluyen en la respuesta.

Entrada	Catálogo con el grafo , Punto migratorio desde el cual inicial las rutas
Salidas	Diccionario con todas las subredes hídricas (componentes), sus estadísticas, sus nodos, grullas, rangos geográficos, orden temporal y tiempo total de ejecución.
Implementado (Sí/No)	Sí se implementó , por todos

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	Complejidad
Paso 1	Verificar existencia del vértice $O(1)$
Paso 2	Recorrer lista de vértices $O(V)$
Paso 3	BFS para cada componente $O(V + E)$
Paso 4	Marcar nodos visitados $O(1)$
Paso 5	Recopilar información de nodos dentro de cada subred $O(K)$
Paso 6	Cálculo de límites geográficos (máximos/mínimos) $O(K)$
Paso 7	Ordenar nodos de cada subred por timestamp $O(K \log K)$
Paso 8	Ordenar subredes por tamaño $O(S \log S)$

TOTAL	$O(V + E + K \log K + S \log S)$
--------------	--

Análisis

El requerimiento se implementó correctamente y permite identificar componentes conectados del grafo hídrico para determinar grupos de individuos potencialmente aislados. El algoritmo recorre eficientemente todos los nodos usando BFS para formar subredes independientes, marca cada vértice visitado para garantizar que cada componente se procese una sola vez y luego analiza cada subred recopilando coordenadas, grullas presentes, eventos y distancias hídricas, además de establecer límites geográficos y un orden temporal para los primeros y últimos registros. Las pruebas realizadas evidenciaron que la solución clasifica correctamente los grupos aislados, produce información detallada de cada subred y mantiene una eficiencia adecuada incluso para grafos grandes, gracias al uso de estructuras hash, listas y recorridos lineales.

Comparación de Resultados Pruebas de ejecución (excel adjunto) con la notación O

REQ 1 – $O(V + E)$

Los tiempos de ejecución crecen de forma casi lineal conforme aumenta el tamaño del grafo. Esto coincide con la complejidad teórica de DFS, donde el recorrido depende directamente del número de vértices y aristas. No se observan crecimientos explosivos, por lo que el comportamiento experimental valida la estimación $O(V + E)$.

REQ 2 – $O(V + E)$

El crecimiento del tiempo es muy similar al del requerimiento 1. BFS presenta un comportamiento estable y proporcional al tamaño del grafo. Las curvas experimentales siguen una tendencia lineal, confirmando que la implementación respeta la complejidad $O(V + E)$.

REQ 3 – $O(V + E + R \log R)$

Los tiempos crecen más que en los dos anteriores debido al ordenamiento de rutas. Experimentalmente se observa una curvatura ligeramente superior a lineal, lo cual concuerda con el término $R \log R$ adicional. La parte dominante sigue siendo el recorrido del grafo, por lo que la complejidad teórica sí se refleja en la ejecución real.

REQ 4 – $O(E \log V)$

Este es uno de los requerimientos más costosos. Los tiempos de ejecución crecen más rápido que en los BFS y DFS, lo cual es coherente con el uso de Prim y colas de prioridad. Experimentalmente se observa crecimiento casi logarítmico sobre el número de aristas, validando claramente el modelo $O(E \log V)$.

REQ 5 – O(E log V)

Presenta un comportamiento muy similar al requerimiento 4, ya que ambos usan Dijkstra/Prim con heap. En las gráficas se observa un crecimiento más acelerado que los BFS, confirmando que el factor logarítmico sí impacta los tiempos reales. La predicción $O(E \log V)$ se cumple correctamente.

REQ 6 – O(V + E + K log K + S log S)

Es el requerimiento más pesado. Experimentalmente es el que presenta mayor tiempo de ejecución. El uso de BFS más ordenamientos internos explica que su crecimiento sea mayor que lineal. Las gráficas reflejan correctamente este aumento, validando el impacto combinado del recorrido del grafo y los ordenamientos logarítmicos.

CONCLUSIÓN

En todos los requerimientos, los tiempos experimentales coinciden con el comportamiento esperado según la Notación Big-O. Los BFS y DFS muestran crecimiento lineal. Prim y Dijkstra muestran crecimiento $E \log V$. Los requerimientos con ordenamientos adicionales presentan crecimiento superior al lineal. No se evidencian desviaciones graves entre teoría y práctica, por lo que las implementaciones por lo menos en tiempo de ejecución serían correctas.