Universidad de los Andes
Facultad de Ingeniería

Departamento de Ingeniería de Sistemas y Computación
Estructuras de Datos y Algoritmos
ISIS-1225

ABET Engineering Accreditation Commission

ANÁLISIS DEL RETO 4

Tomás Sierra Sanchez, 202221567, t.sierras@uniandes.edu.co

Juan Francisco Rodríguez, 202214603, jf.rodriguezc1@uniandes.edu.co

Carlos Peña, 202222516, c.penaa@uniandes.edu.co

Requerimiento 1

Descripción

Este requerimiento se encarga de encontrar un camino entre dos puntos de encuentro

Entrada	Control: El catálogo de datos completo, Identificador_origen: El identificador de donde se va a iniciar la búsqueda, Identificador_destino: El identificador de donde se va a finalizar la búsqueda
Salidas	- La Distancia total de recorrido - El total de puntos de encuentro que contiene el camino - El total de nodos de seguimiento - Una tabla con los 5 primeros y los 5 últimos vértices
Implementado (Sí/No)	Si. Tomás Sierra

Análisis de complejidad

Pasos	**Complejidad**
Ejecutar DFS en el grafo	O(V + E)
Buscar el camino	O(1)
Recorrer cada vértice	O(v)
Añadir a la lista final	O(n)
TOTAL	***O(V+E) + O(v*n)***

Pruebas Realizadas

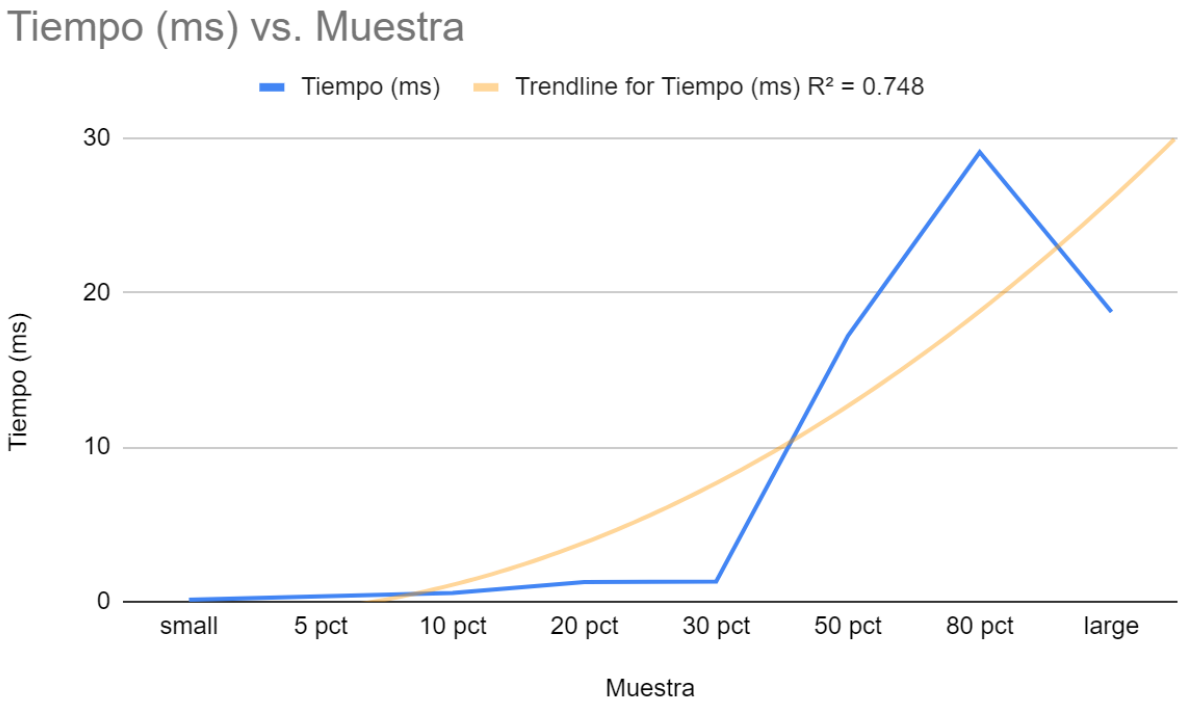
Los parámetros de entrada son: m111p862_57p449 y m111p908_57p427. Las pruebas realizadas fueron realizadas en una máquina con las siguientes especificaciones.

Procesadores	**AMD Ryzen 7 5700G 3.80 GHz (8 cores 16 threads)**
Memoria RAM	32 GB 3600 MHz DDR4
Sistema Operativo	Windows 10 Home - 21H2

Tablas de datos

Muestra	Tiempo (ms)
small	0.12
5 pct	0.34
10 pct	0.56
20 pct	1.27
30 pct	1.31
50 pct	17.2
80 pct	29.08
large	5.49

Gráficas



Análisis

Podemos observar que en la práctica la curva que más se asemeja a esta gráfica es una complejidad de O(n^2) sin embargo no se asemeja por mucho. Esto se puede deber a dos factores importantes. Primero la complejidad mayor de todo el procedimiento es de O(V+E) por el recorrido DFS, sin embargo cuando se piensa que el grafo cada vez es más grande y más complejo esta complejidad va a crecer exponencialmente. Luego podemos ver como DFS devuelve un camino, no necesariamente el más corto por lo que si uno corre con mala suerte el algoritmo podría elegir un camino bastante largo como lo pudo ser el caso de 80%. Y es por eso que se muestra esta complejidad tan compleja.

Requerimiento 2

Descripción

Este requerimiento se encarga de encontrar el camino más corto en términos de vértices entre dos puntos de encuentro.

Entrada	Control: El catálogo de datos completo, Identificador_origen: El identificador de donde se va a iniciar la búsqueda, Identificador_destino: El identificador de donde se va a finalizar la búsqueda
Salidas	- La Distancia total de recorrido - El total de puntos de encuentro que contiene el camino - El total de nodos de seguimiento - Una tabla con los 5 primeros y los 5 últimos vértices
Implementado (Sí/No)	Si. Tomás Sierra

Análisis de complejidad

Pasos	**Complejidad**
Ejecutar DFS en el grafo	O(V + E)
Buscar el camino	O(1)
Recorrer cada vértice	O(v)
Añadir a la lista final	O(n)
TOTAL	***O(V+E) + O(v*n)***

Pruebas Realizadas

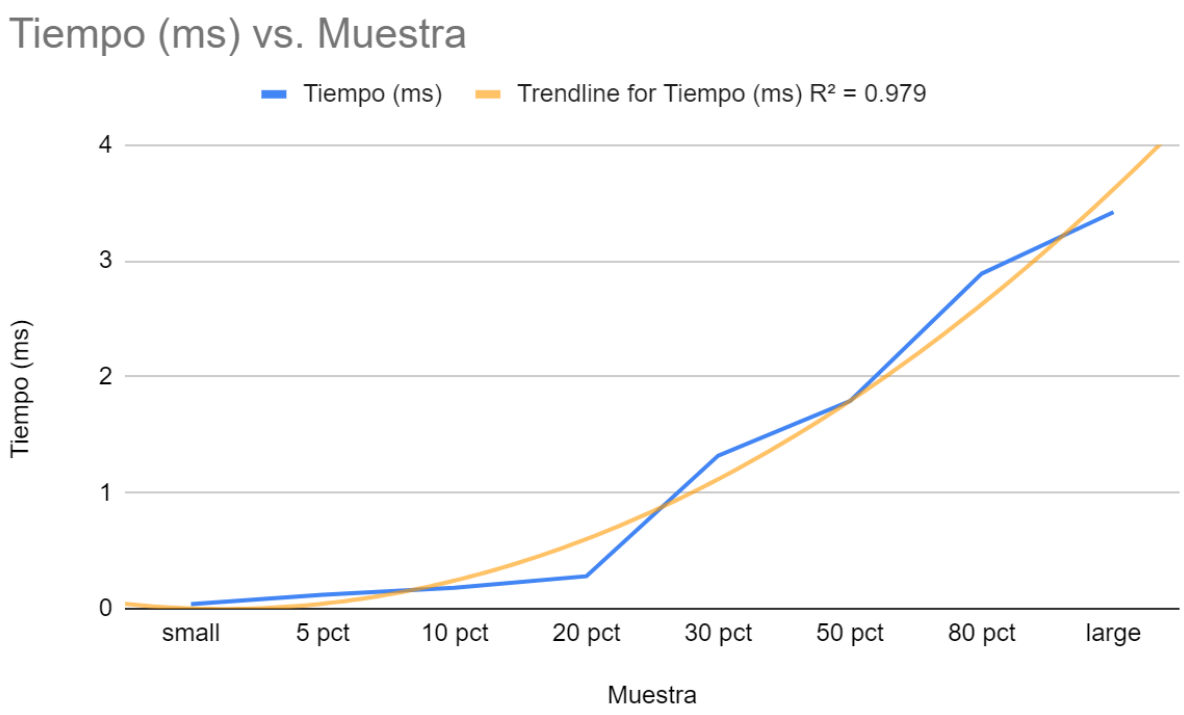
Los parámetros de entrada son: m111p862_57p449 y m111p908_57p427. Las pruebas realizadas fueron realizadas en una máquina con las siguientes especificaciones.

Procesadores	**Intel(R) Core(TM) i7-1165G7 @ 2.80GHz 2.80 GHz**
Memoria RAM	8 GB
Sistema Operativo	Windows 11 Single Language

Tablas de datos

Muestra	Tiempo (ms)
small	0.04
5 pct	0.12
10 pct	0.18
20 pct	0.28
30 pct	1.32
50 pct	1.79
80 pct	2.89
large	3.42

Gráficas



Análisis

Se puede observar que en la práctica la implementación tiene una complejidad de O(n^2) esto tiene sentido ya que cuando se aplica BFS similar a DFS se suman los vértices más los arcos pero si estos dos números tienen una relación llena lo que resulta que el algoritmo tenga una complejidad O(v^2) eso demuestra la gráfica.

Requerimiento 3

Descripción

Este requerimiento se encarga de encontrar el comportamiento de todas las manadas de lobos en el estudio.

Entrada	Catálogo con todos los datos
Salidas	- El total de manadas encontradas - Las 5 manadas con mayor dominio en el territorio
Implementado (Sí/No)	Sí, Tomás Sierra

Análisis de complejidad

Pasos	**Complejidad**
Realizar kosaraju	O(V+E)
Iterar sobre las llaves del mapa	O(L)
Iterar sobre las manadas	O(M)
Iterar por cada lobo de la manada	O(lo)
TOTAL	***O(V+E) + O(L) + O(M*lo)***

Pruebas Realizadas

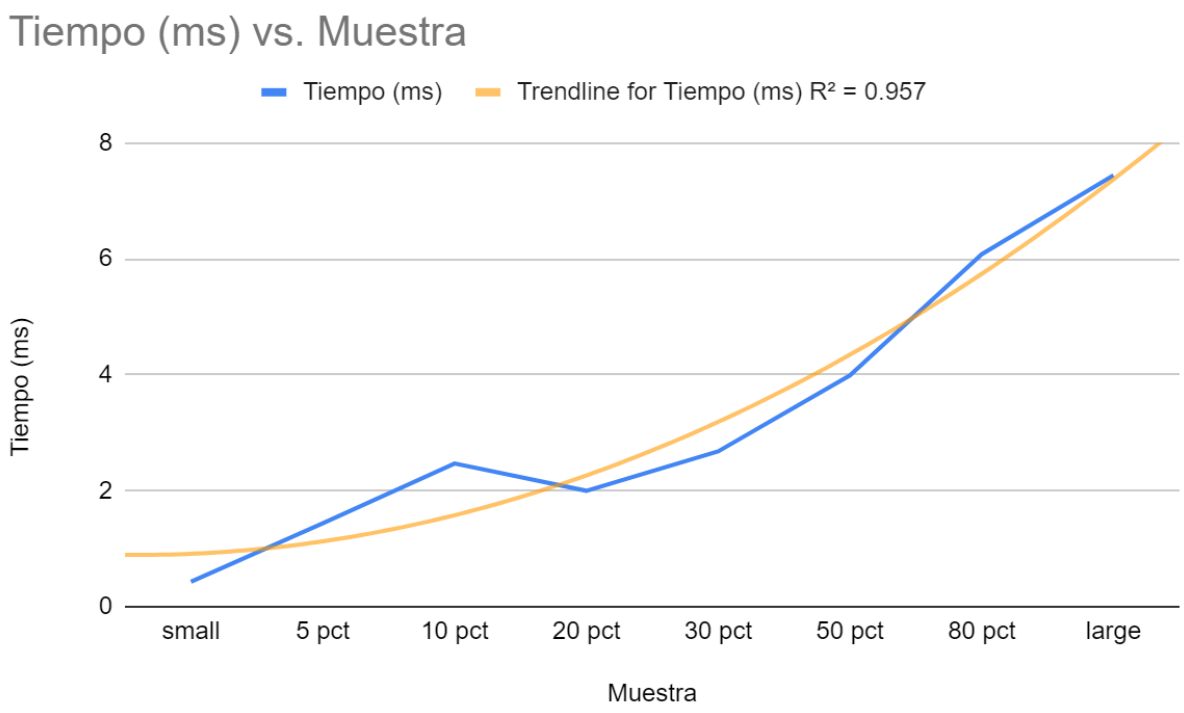
Las pruebas realizadas fueron realizadas en una máquina con las siguientes especificaciones.

Procesadores	**AMD Ryzen 7 5700G 3.80 GHz (8 cores 16 threads)**
Memoria RAM	32 GB 3600 MHz DDR4
Sistema Operativo	Windows 10 Home - 21H2

Tablas de datos

Muestra	**Tiempo (ms)**
small	0.43
5 pct	1.44
10 pct	2.47
20 pct	2.00
30 pct	2.68
50 pct	3.99
80 pct	6.08
large	7.44

Gráficas



Análisis

Se puede observar que la curva se apega a un comportamiento cuadrático O(n^2) esto tiene sentido ya que en teoría lo que más tiempo va a necesitar va a ser el doble recorrido de las listas de manadas y de lobos completamente. Si estas dos listas tuvieran la misma cantidad de elementos g entonces hacerlo dos veces sería g^2 que es más o menos lo que se identifica en la gráfica.

Requerimiento 4

Descripción

Este requerimiento se encarga de

Entrada	El catálogo de datos completo, punto de origen y punto de destino
Salidas	Distancia entre puntos, total de puntos de encuentro, lobos que usan el corredor, camino más rápido, primeros y últimos 3 puntos de encuentro.
Implementado (Sí/No)	Si, Juan Francisco Rodríguez Contreras

Análisis de complejidad

Pasos	Complejidad

Realizar el algoritmo de Dijkstra	O(V + E log E)
Iterar sobre grafo	O(G)
Iterar sobre las puntos de encuentro	O(P)
Iterar por cada lobo	O(L)
TOTAL	O(V + E log E) + O(G) + O(P) + O(L)

Pruebas Realizadas

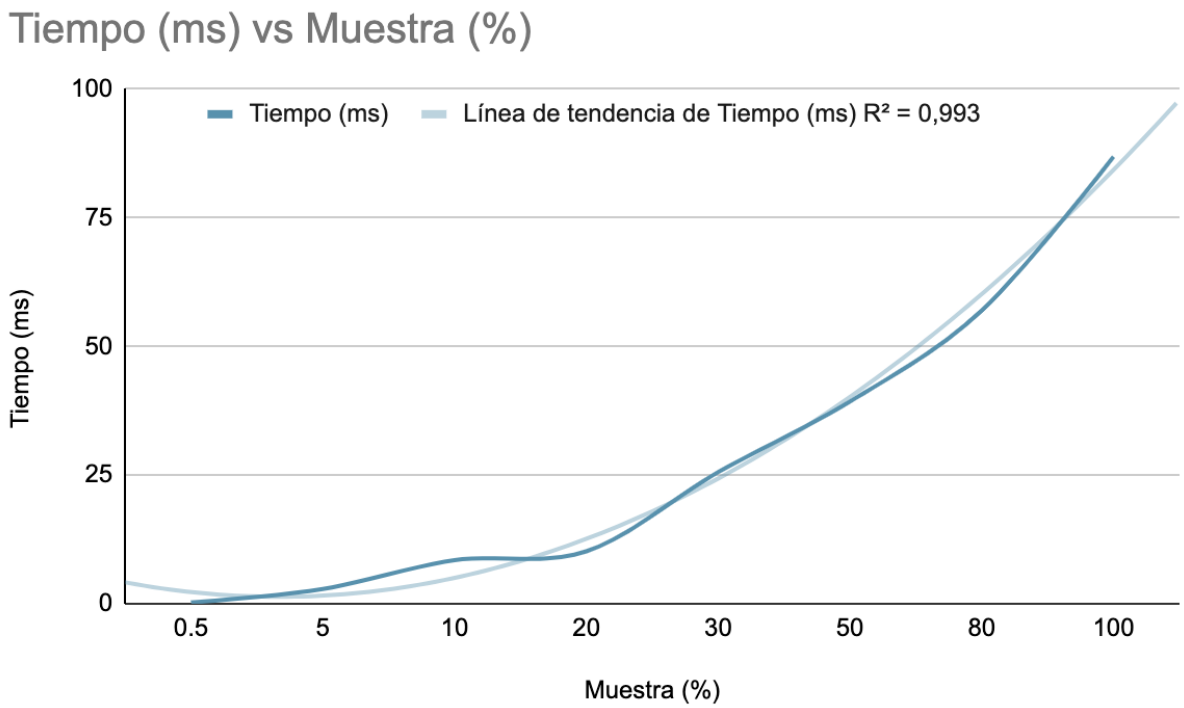
Los parámetros de entrada fueron punto de origen (-111.911, 57.431) y punto de destino (-111.865, 57.435). Las pruebas realizadas fueron realizadas en una máquina con las siguientes especificaciones.

Procesadores	1,8 GHz Intel Core i5 de dos núcleos
Memoria RAM	8 GB 1600 MHz DDR3
Sistema Operativo	MacOS Catalina Versión 10.15.7

Tablas de datos

Muestra	Tiempo (ms)
small	0,25
5 pct	2,86
10 pct	8,48
20 pct	10,18
30 pct	25,57
50 pct	39,3
80 pct	56,99
large	86,87

Gráficas



Análisis

Viendo el resultado que nos arroja la gráfica se puede notar una tendencia cuadrática debido a que debemos encontrar el camino más corto entre dos puntos de un hábitat y gracias a esto debemos utilizar el algoritmo de Dijkstra por el cual se nota que prevalece un comportamiento cuadrático a la hora de realizar pruebas.

Requerimiento 5

Descripción

Este requerimiento se encarga de reconocer el corredor migratorio más extenso.

Entrada	Control: El catálogo de datos completo, o: punto de encuentro de origen, d: Distancia que puede recorrer el guardabosques desde el punto de origen [km], m: El número mínimo de puntos de encuentros que el guardabosques desea inspeccionar
Salidas	# de posibles rutas y la mejor teniendo en cuenta distancia recorrida y puntos visitados.
Implementado (Sí/No)	Si, Carlos Peña

Análisis de complejidad

Pasos	Complejidad
Crear una estructura MST con el algoritmo Prim en su versión eager del grafo no dirigido. Donde se analizan sus vértices (V) y sus arcos (E) para construir el mismo.	O(E log(V))
Para cada punto en los adyacentes (a) al punto de encuentro inicial, se analiza un camino usando la función auxiliar para: 1. 'Puntos de encuentro y seguimiento visitados' 2. 'Distancia recorrida [km]' 3. 'Lista de puntos' 4. 'Secuencia de posibles individuos en el trayecto' Posteriormente se guarda en un arraylist 'paths'	O(a)
Se crean las variables auxiliares c (para contar el número de caminos con distancia mayor que la del recorrido que puede dar el guardabosques) y path_points para guardar los puntos que ya fueron analizados del mapa. Mientras que el número de veces que se genera un camino que no se puede recorrer, sea menor a el número de caminos ya recorridos + 2 (pa). - Esto se debe a que si se piensa en los caminos como un árbol binario, nos damos cuenta que siempre el número de posibles nuevos arcos, va a ser igual al número de caminos ya recorridos + 2. Tome por ejemplo un árbol con 7 elementos de 2 niveles completos. Tiene 8 posibles nuevas ramas (nuevos caminos) y tenemos un total de 6 ramas guardadas (caminos). - De esta forma nos aseguramos que cuando en verdad no existan más variaciones para los caminos, todas las esquinas (ramas exteriores) hayan sido recorridas. Para cada punto en las llaves del mapa 'edgeTo' (p), si va (vértice inicial) está en los puntos visitados ('path_points') y vb (vértice destino) no está en los puntos visitados (path_points). Se obtienen los valores específicos (para la función auxiliar) del camino que debe tomar para poder añadir este punto a la lista de puntos. Esto, al recorrer todos los posibles caminos (paths) y encontrar el indicado. Por ej, para va = C y vb = D (C → D) un camino **no** indicado sería A	O(pa * p * paths)

→ B, mientras que uno indicado sería A → C para completar el camino A → C → B. Después de encontrar este camino, el mismo es actualizado con uso de la función auxiliar, y se guarda en 'paths'. Adicionalmente, vb se añade a la lista 'path_points'. Finalmente, si la distancia recorrida es menor a el recorrido que puede dar el guardabosques y la longitud de la lista de puntos es mayor a 2, se suma a la lista 'paths' este nuevo camino y se resta 1 al contador (y el número de adyacentes - 1 si el siguiente punto, es un punto de encuentro). De lo contrario, se suma a el contador 1, señalando que se creó un camino imposible.	
Posteriormente, ordenamos estos caminos ('paths'). Primero por el criterio de número de puntos visitados, y después por la distancia recorrida mediante mergesort. Obteniendo así una lista con un orden de distancia descendente; y de ser esta la misma, un número de puntos visitados descendente.	O(2 paths log(paths)) ≈ O(paths log(paths))
A continuación filtramos nuestro arraylist de 'paths' dado que tenemos varios caminos repetidos. Por ej, los caminos A → B y A → B → C en realidad son solo 1 camino válido, A → B → C. Para esto recorremos todos los posibles caminos ('paths') y para cada uno vemos si su 'Lista de puntos' es subset de otro camino. De ser así, significa que es un path a eliminar y se guarda en la lista 'del_list'. (A → B es subset de A → B → C)	O(paths * paths)
Finalmente, aplicamos el filtro a nuestro arraylist de 'paths' ordenados. Adicionalmente, chequeamos para caminos que no cumplan con el requisito de tener un número mayor o igual de puntos visitados a el dado por parámetro. Los caminos que cumplen todos los anteriores son guardados en el arraylist 'newpaths' el cual es retornado.	O(paths)
TOTAL	***O(E log(V)) + O(a) + O(pa * p * paths) + O(paths log(paths)) + O(paths * paths) + O(paths)***

Pruebas Realizadas

Los parámetros de entrada son: catálogo, o: m112p039_56p612, d: 24.5, m: 2. Las pruebas realizadas fueron realizadas en una máquina con las siguientes especificaciones:

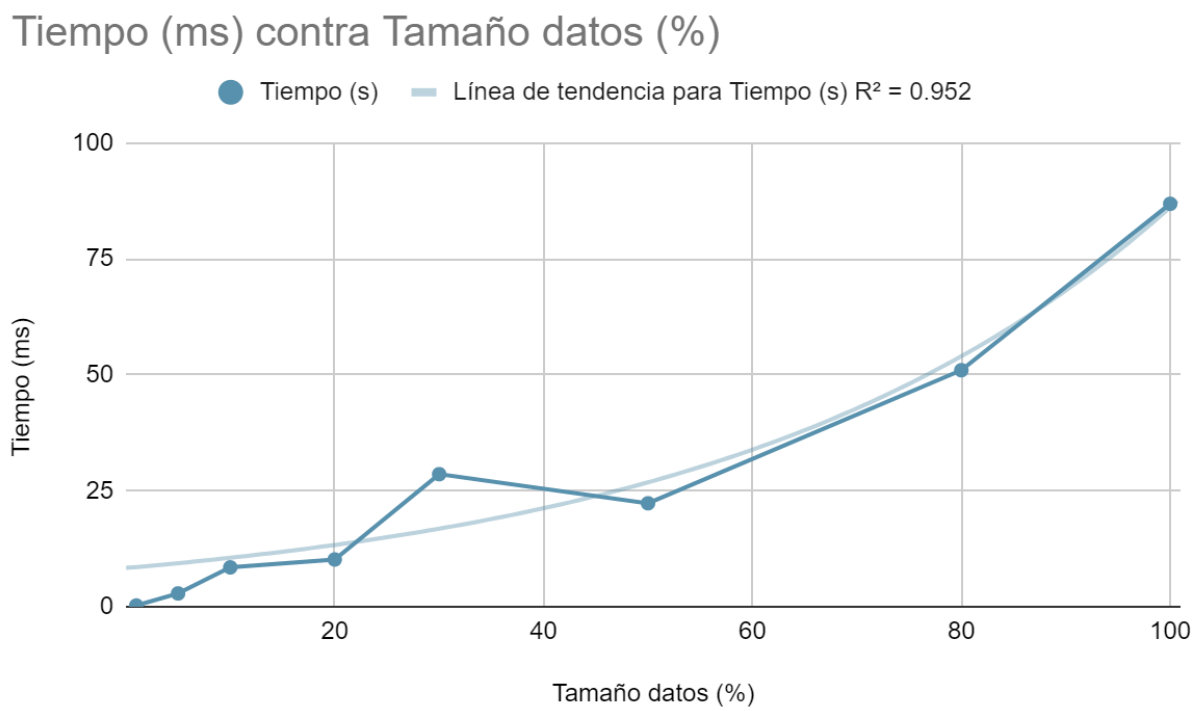
Procesadores	**AMD Ryzen 7 5700G 3.80 GHz (8 cores 16 threads)**
Memoria RAM	32 GB 3600 MHz DDR4

Sistema Operativo	Windows 10 Home - 21H2

Tablas de datos

Tamaño datos (%)	Tiempo (ms)
1	0.25
5	2.86
10	8.48
20	10.18
30	28.57
50	22.3
80	50.99
100	86.87

Gráficas



Análisis

Se puede observar que en la práctica el requerimiento tiene una complejidad aproximada O(E log(V)) + O(pa * p * paths). Esto se debe a que la complejidad que más aporta al tiempo es la creación del MST del

grafo y construir los caminos en base a el MST. Pues filtrar y ordenar los caminos, no requieren de tantos recursos como el algoritmo de Prim (versión Eager) y la construcción de cada camino posible desde el punto de encuentro de inicio a vértices adyacentes.

Requerimiento 6

Descripción

Este requerimiento se encarga de Identificar diferencias en los corredores migratorios

según el tipo de individuo

Entrada	Control: El catálogo de datos completo, fi: fecha inicial, ff: fecha final, g: género.
Salidas	Lobo que más y menos recorrió
Implementado (Sí/No)	Si, Carlos Peña

Análisis de complejidad

Pasos	**Complejidad**
Para cada lobo (l) en el mapa 'map' y para cada registro (r) de cada lobo, se chequea que cumpla con los requisitos del requerimiento, tiempo y género. Si cumple los anteriores se guarda la información pertinente y la suma de los pesos de los arcos que conectan los puntos. Al final se construye la información pertinente de todos los puntos del camino del lobo (pv) si la distancia recorrida es mayor a 0.	O(l * r * pv)
Finalmente se ordenan el número de lobos válidos (lv) con mergesort.	O(lv log(lv))
TOTAL	***O(l * r * pv) + O(lv log(lv))***

Pruebas Realizadas

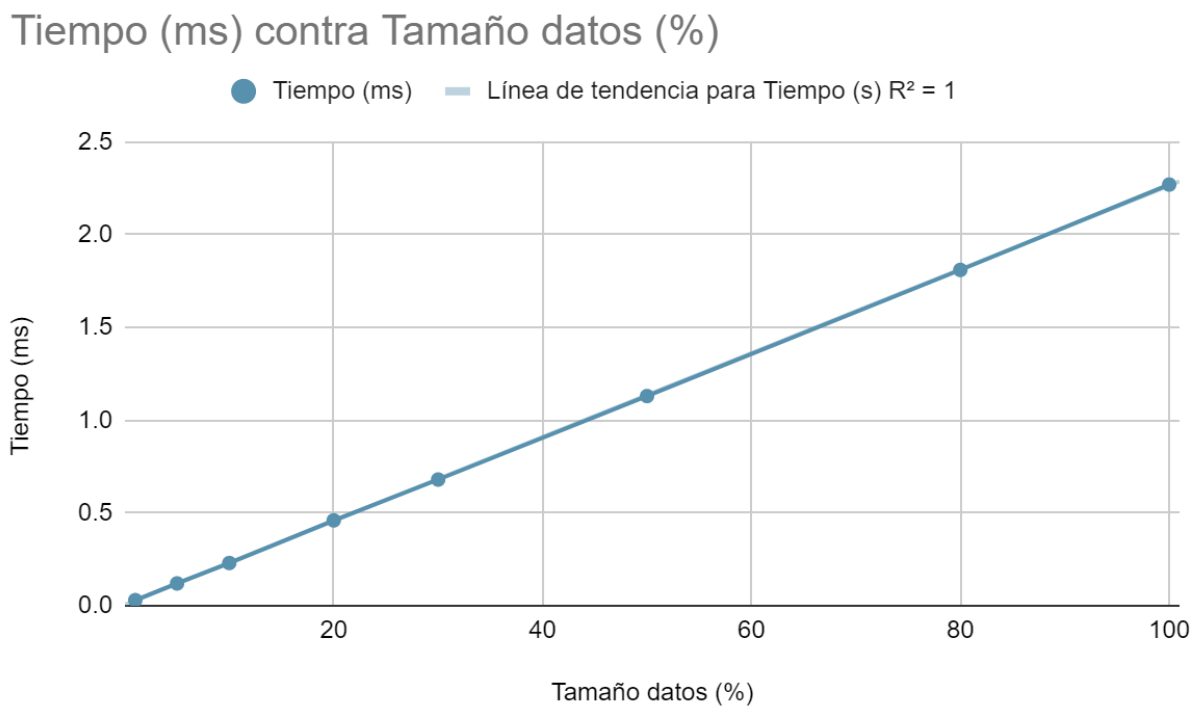
Los parámetros de entrada son: catálogo, fi: 2013-02-16 00:00:00 ff: 2014-10-23 23:59:00 g: f. Las pruebas realizadas fueron realizadas en una máquina con las siguientes especificaciones:

Procesadores	**AMD Ryzen 7 5700G 3.80 GHz (8 cores 16 threads)**
Memoria RAM	32 GB 3600 MHz DDR4
Sistema Operativo	Windows 10 Home - 21H2

Tablas de datos

Tamaño datos (%)	Tiempo (ms)
1	0.03
5	0.12
10	0.23
20	0.46
30	0.68
50	1.13
80	1.81
100	2.27

Gráficas



Análisis

En este caso podemos observar como la complejidad tiene un comportamiento lineal. Esto se debe a que se parece en cierto modo a la carga de datos, solo que filtramos los datos a procesar. Adicionalmente el

mergesort no es tan significativo, pues no es organizar un número significativo de elementos máximo serán la cantidad de lobos de dicho género (en el peor caso 46).

Requerimiento 7

Descripción

Este requerimiento se encarga de encontrar el comportamiento de las manadas dependiendo de unas condiciones externas.

Entrada	- Control: El catálogo de datos completo - Fecha Inicial - Fecha Final - Temperatura Inicial - Temperatura Final
Salidas	- Una tabla con las 5 manadas con el mayor dominio bajo los parámetros de entrada - Un mapa con las manadas.
Implementado (Sí/No)	Si, Carlos Peña y Tomás Sierra

Análisis de complejidad

Pasos	**Complejidad**
Crear el grafo de nuevo se realiza lo mismo que la carga de datos sin los sorts ya que ya está ordenado	*O(nt) + O(l * e)+ O(ni *k)*
Realizar kosaraju	O(V+E)
Iterar sobre las llaves del mapa	O(L)
Iterar sobre las manadas	O(M)
Iterar por cada lobo de la manada	O(lo)
TOTAL	***O(nt) + O(l * e)+ O(ni *k) + O(Lobos * Lista Lobos) + O(Nueva Lista) + O(Lista Pos) + O(V+E) + O(L) + O(M*lo)***

Pruebas Realizadas

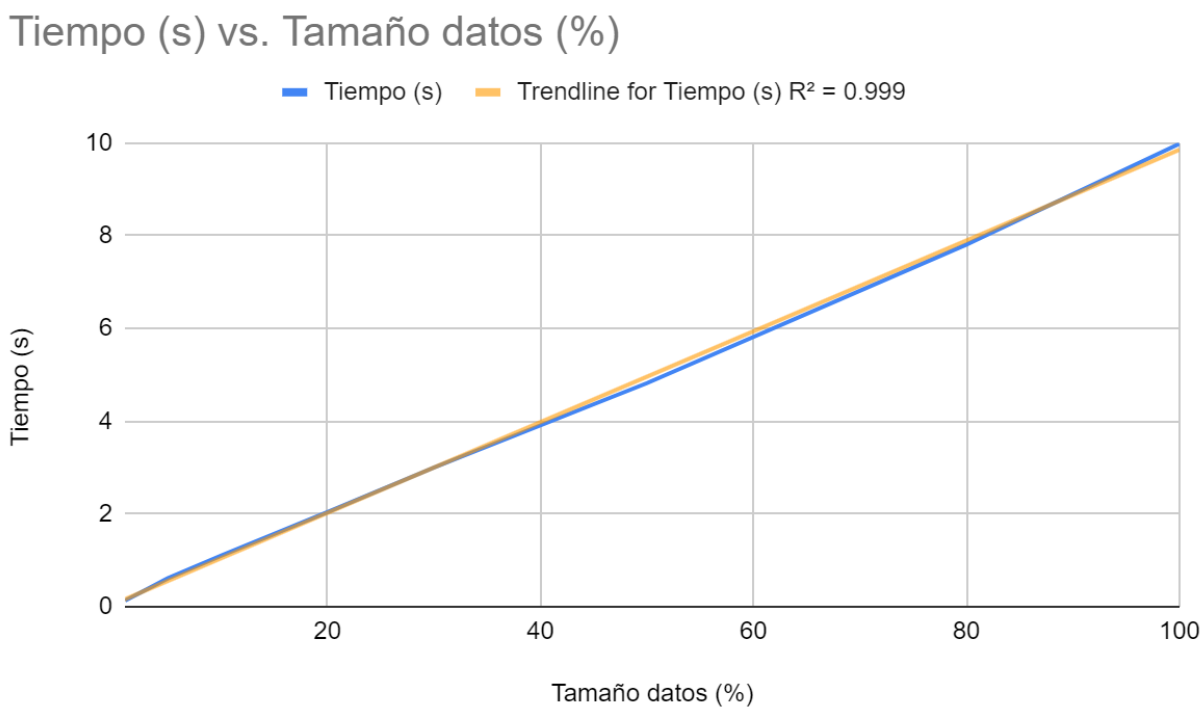
Los parámetros de entrada son: 2012-11-28 00:00:00, 2014-05-17 23:59:00, -17.3, 9.3 y el catálogo. Las pruebas realizadas fueron realizadas en una máquina con las siguientes especificaciones:

Procesadores	**AMD Ryzen 7 5700G 3.80 GHz (8 cores 16 threads)**
Memoria RAM	32 GB 3600 MHz DDR4
Sistema Operativo	Windows 10 Home - 21H2

Tablas de datos

Tamaño datos (%)	Tiempo (s)
1	0.13
5	0.61
10	1.1
20	2.04
30	3
50	4.82
80	7.81
100	9.98

Gráficas



Análisis

Se puede observar que el requerimiento tiene una complejidad de O(n), esto es muy bueno y además tiene sentido ya que en primer lugar estamos realizando un procedimiento muy similar al de la carga de datos pero le quitamos la peor complejidad que es la de realizar el merge sort. También para Kosaraju la complejidad se mantiene constante y ayuda a que tanto la complejidad esperada como la demostrada sea lineal.

Carga de datos

Descripción

La carga de datos del reto.

Entrada	Control: El catálogo, filename: Nombre del archivo a cargar
Salidas	Control con grafo dirigido y no dirigido, mapa con eventos de cada lobo ordenados cronológicamente, mapa con especificaciones de cada lobo, diccionario con puntos de encuentro y lobos presentes en los mismos.
Implementado (Sí/No)	Si, Carlos Peña

Análisis de complejidad

Pasos	Complejidad
Para cada línea en el archivo 'tracks' (nt) se genera un id único, se añade al mapa de eventos 'map'. En este el ID de cada lobo es la llave y el valor es un arraylist con los eventos. Se actualizan los valores de lat-long mín. y máx. Al finalizar se obtiene paralelamente el valor de eventos analizados para el print.	O(nt)
Ahora agrupamos la información en el mapa. 1. Se recorre cada lobo (l) en el mapa de eventos y se obtiene su lista de eventos. 2. Se ordena con mergesort los eventos (e) del lobo por orden cronológico del más viejo al más reciente. Se añade la lista ordenada a un nuevo mapa ordenado.	O(l * e log(e))
Paralelamente con la agrupación se van construyendo los grafos de la siguiente manera: 1. Para cada evento (e) de la lista ordenada de cada lobo (l), se genera a. Una llave para el punto de seguimiento b. Una llave tipo punto de encuentro c. ID del lobo 2. Se verifica si este evento es el primero	O(l * e)

3. Si no lo es, se verifica que este no sea el mismo que el anterior y que no sea el último en los puntos recorridos previamente por el lobo 4. Se insertan los vértices y arcos correspondientes identificando si: a. Crea un punto de encuentro b. Es parte de un punto de encuentro previo c. Ya existen arcos que conectan dichos vértices 5. Se actualiza con cada iteración el valor de la anterior llave y de puntos de encuentro del grafo y los individuos en los mismos. Finalmente se actualiza el valor del mapa de eventos con el ordenado y se añade el diccionario de puntos de encuentro 'pedict'.	
Se carga el archivo 'individuals' (ni), para cada llave (k) del mismo se verifica si este tiene un valor nulo y se reemplaza con None. Posteriormente se genera un ID de lobo y se añade la información de especificaciones a un mapa 'map_spec'.	O(ni * k)
TOTAL	***O(nt) +O(l * e log(e)) + O(l * e) O(ni *k)***

Pruebas Realizadas

Las pruebas realizadas fueron realizadas en una máquina con las siguientes especificaciones:

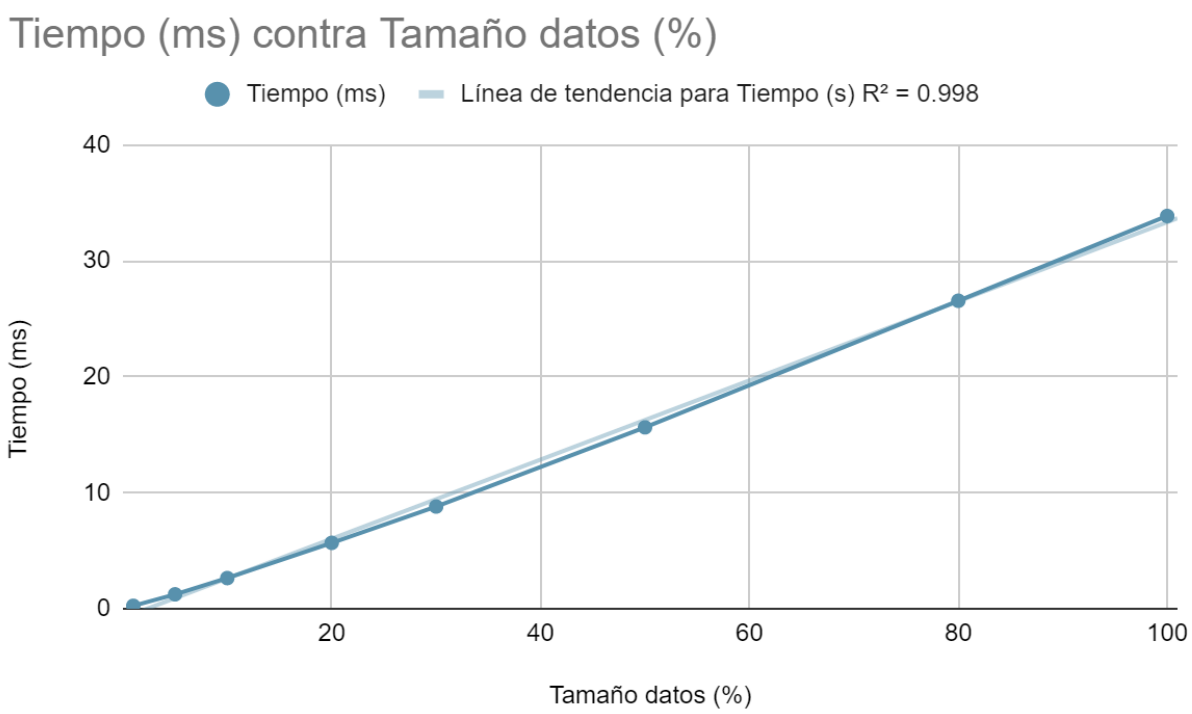
Procesadores	**AMD Ryzen 7 5700G 3.80 GHz (8 cores 16 threads)**
Memoria RAM	32 GB 3600 MHz DDR4
Sistema Operativo	Windows 10 Home - 21H2

Tablas de datos

Tamaño datos (%)	Tiempo (ms)
1	0.25
5	1.24
10	2.64
20	5.68
30	8.81
50	15.64

80	26.56
100	33.87

Gráficas



Análisis

Se puede observar que en la práctica el requerimiento tiene una complejidad aproximada O(nt). Esto se debe a que la complejidad que más aporta al tiempo es el recorrer los datos de seguimiento. Pues ordenarlos no requiere de tantos recursos como recorrerlos y agruparlos en este caso.