Programación Avanzada y Estructura de Datos

Proyecto del Segundo Semestre – Estructuras No Lineales

Los Caballeros de la Mesa de Hash

equipo = {
"Nombre del equipo": "Grupo 11",
"Miembro 1": "Carlos Romero (c.romero)",
"Miembro 2": "Marc González (marc.gcarro)",
"Miembro 3": "Javier Iván (javierivan.carceles)",
"Miembro 4": "Pablo Molina (p.molina)"
}

La Salle - Universitat Ramon Llull

21 mayo de 2023

Índice

1. Lenguaje de programación escogido

Java es un lenguaje de programación de alto nivel, orientado a objetos y de propósito general que ha sido ampliamente adoptado en la industria y la academia. Sus principales ventajas incluyen:

1.1. Ventajas

- **Compatibilidad multiplataforma:** Gracias a la Máquina Virtual Java (JVM), el código Java puede ejecutarse en cualquier plataforma compatible, lo que facilita el desarrollo de aplicaciones que se ejecuten en diversos sistemas operativos sin la necesidad de realizar modificaciones adicionales.
- **Bibliotecas y ecosistema:** Java cuenta con un amplio conjunto de bibliotecas y marcos de trabajo que facilitan la implementación de funcionalidades en áreas como el desarrollo web, móvil, inteligencia artificial, machine learning, big data y computación en la nube. El ecosistema de Java también incluye numerosos recursos y herramientas de desarrollo, como el entorno de desarrollo integrado (IDE) IntelliJ IDEA, que optimiza la productividad de los programadores.
- **Seguridad y confiabilidad:** Java ha sido diseñado con características de seguridad robustas, como la gestión de memoria automática y la prevención de acceso no autorizado a recursos del sistema. Estas características son fundamentales en aplicaciones críticas y empresariales, donde la protección de datos y la integridad del sistema son esenciales.
- **Comunidad y soporte:** Java cuenta con una gran comunidad global de desarrolladores, lo que asegura un flujo constante de innovaciones, soluciones y mejoras en el lenguaje. Además, hay una amplia variedad de recursos de aprendizaje y documentación disponibles para resolver problemas y encontrar soluciones ya implementadas por otros profesionales.

1.2. Desventajas

A pesar de estas ventajas, también se deben considerar algunos aspectos que podrían mejorarse en Java:

- **Sintaxis y complejidad:** Java tiene una sintaxis más detallada y rigurosa en comparación con otros lenguajes modernos como Python. Esto puede generar una curva de aprendizaje más pronunciada y, en ciertos casos, una menor productividad en el desarrollo.
- **Rendimiento y eficiencia:** Java puede ser menos eficiente en términos de rendimiento que lenguajes compilados directamente al código de máquina, como C o C++. La dependencia de la JVM puede generar una sobrecarga adicional en la ejecución del código.
- **Cambios y evolución en el lenguaje:** Java ha experimentado numerosas actualizaciones y cambios a lo largo de su historia, lo que puede generar problemas de compatibilidad entre versiones y un desafío en la adopción de las últimas características del lenguaje.

Generalmente, Java es un lenguaje de programación versátil y sólido que ofrece ventajas significativas en términos de compatibilidad multiplataforma, bibliotecas y ecosistema,

seguridad y soporte comunitario. Sin embargo, también presenta desafíos en cuanto a sintaxis, rendimiento y evolución del lenguaje, lo que debe ser considerado al elegirlo para proyectos.

2. Estructuras de datos no lineales

Durante este curso, hemos estudiado en profundidad diversas estructuras de datos no lineales, las cuales juegan un papel fundamental en la resolución de problemas computacionales complejos y en la optimización del rendimiento en el ámbito del almacenamiento y recuperación de información. En este informe, presentaremos una introducción a cuatro estructuras de datos no lineales clave que hemos estudiado: grafos, árboles de búsqueda binarios auto-balanceados (AVL Trees), R-Trees y tablas de hash. Analizaremos sus características principales, aplicaciones y ventajas en comparación con otras estructuras de datos.

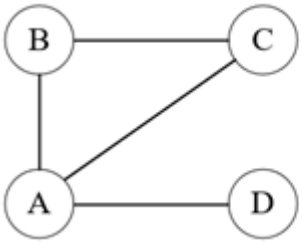
2.1. Grafos

2.1.1. Introducción

Los grafos son estructuras de datos que consisten en un conjunto de nodos (también llamados vértices) y aristas (conexiones entre los nodos). Los grafos pueden ser dirigidos (las aristas tienen una dirección específica) o no dirigidos (las aristas no tienen dirección). Estas estructuras son especialmente útiles para representar relaciones entre elementos en sistemas complejos y se utilizan en áreas como el análisis de redes sociales, algoritmos de rutas en sistemas de navegación y en la resolución de problemas de optimización, como el problema del viajante de comercio.

2.1.2. Diseño

Para representar toda la información que leamos de los ficheros de texto, creamos una matriz adyacente con la siguiente estructura, ver Figura 1.



	A	B	C	D
A	0	1	1	1
B	1	0	1	0
C	1	1	0	0
D	1	0	0	0

Ilustración 1 Representación de nodos en una matriz de adyacencia

Siendo A, B, C, etc. los ID's de los puntos de interés en la matriz de adyacencia, consideramos eliminar la información de la diagonal superior debido a que, al tratarse de una matriz simétrica, la información se encuentra duplicada. Sin embargo, tras consultar con el profesor y analizar su opinión, concluimos que, en términos de optimización, no sería beneficioso. Aunque sí ahorraríamos espacio en la memoria, perderíamos rendimiento, ya que tendríamos que eliminar información previamente añadida. Por lo tanto, esta estrategia no resulta útil.

2.1.3. Algoritmos implementados

Este parte del proyecto tiene como objetivo la implementación de un sistema para optimizar las rutas de mensajería aérea en el contexto medieval, utilizando aves mensajeras, especialmente

las golondrinas europeas y africanas. El sistema se basa en un archivo[XXS-XXL] .paed que contiene datos sobre lugares de interés y rutas conocidas entre ellos.

Las funcionalidades para implementar son:

- **Representación como gráfico:** El código deberá definir e implementar la representación de un gráfico para almacenar los datos proporcionados de manera eficiente.

- **Exploración del reino:** Esta función permite al usuario seleccionar un lugar de interés y muestra todos los lugares de interés en el mismo reino que se pueden alcanzar desde el lugar seleccionado, priorizando aquellos a los que se puede llegar sin salir del reino. Utiliza el algoritmo de búsqueda en anchura (BFS) para recorrer el grafo y encontrar los lugares de interés alcanzables. La función `bfs` implementa el algoritmo BFS y utiliza una matriz de adyacencia `adjMatrix` para representar las conexiones entre los lugares de interés. Muestra los lugares de interés en el mismo reino y calcula el tiempo de ejecución.

- **Detección de rutas habituales:** Esta función intenta encontrar las rutas más comunes en el sector de la mensajería aérea, minimizando la distancia total a recorrer. Se asume que las aves vuelan en línea recta. Utiliza el algoritmo del árbol de expansión mínimo (MST) para encontrar el conjunto de rutas que conectan todos los lugares de interés. La función `detectHabitualPaths` implementa el algoritmo MST y muestra las rutas más habituales, calculando el tiempo de ejecución.

- **Mensajería premium:** Esta función calcula la forma más rápida de enviar un mensaje desde un lugar de interés a otro, considerando si la golondrina está cargando un coco y si sería más eficiente enviar una golondrina europea o africana. Utiliza el algoritmo de Dijkstra para encontrar el camino más corto entre los lugares de interés. La función `calculateShortestPath` implementa el algoritmo de Dijkstra y devuelve el camino más corto. La función `premiumMessaging` recopila la información necesaria del usuario, como el origen, el destino y si la golondrina lleva un coco, y muestra el resultado con la opción más eficiente, calculando el tiempo y la distancia. También muestra el camino recorrido por la golondrina.

En general, estos algoritmos son útiles para diferentes situaciones en las que se necesita encontrar rutas, explorar conexiones o calcular caminos más cortos en un grafo. En el contexto del código proporcionado, estos algoritmos se utilizan para diversas funcionalidades relacionadas con la gestión de lugares de interés y la mensajería aérea.

BFS

El algoritmo de búsqueda de amplitud (BFS) es un algoritmo utilizado para recorrer o explorar estructuras de datos en forma de gráficos o árboles. En este caso, se utiliza para descubrir un "dominio" representado por la matriz de adyacencia. El objetivo es encontrar todos los lugares de interés a los que se puede llegar desde un punto de partida determinado.

La función “exploration()” es el punto de entrada del algoritmo. Comienza pidiéndole al usuario que ingrese la identificación del lugar de interés que desea explorar. Luego se realiza una

búsqueda en la lista de puntos de interés para encontrar la posición en la tabla correspondiente al punto ingresado. Si el punto de interés no se encuentra en el sistema, se mostrará un mensaje de error. De lo contrario, se muestra información sobre el lugar de interés y se llama a la función bfs para realizar el análisis.

La función bfs implementa el algoritmo de búsqueda en amplitud. Comienza creando una cola vacía utilizando la interfaz Queue de Java, que le permite agregar elementos al final de la cola y eliminar elementos de la parte superior de la cola. También se crea una matriz de valores booleanos llamados visitados, que se utiliza para realizar un seguimiento de los lugares favoritos visitados durante el descubrimiento. Inicialmente, todos los elementos de visitados se configuran como falsos, lo que indica que aún no se han visitado lugares favoritos.

El algoritmo comienza marcando el punto de interés inicial como visitado y agregándolo a la cola. Esto se hace configurando el elemento correspondiente en la matriz visitada como verdadero y usando el método add de la cola para agregar el índice del lugar de interés original.

A continuación, se inicia un bucle principal y se ejecuta siempre que la cola no esté vacía. En este bucle, se recupera un elemento de la cola mediante el método poll, que devuelve y elimina el elemento de la parte superior de la cola. Este elemento representa el punto de interés que se está descubriendo.

Si el punto de interés actual pertenece al mismo dominio que el punto original, se mostrará información al respecto. Esto se hace comprobando si el valor del atributo "reino" del punto de interés actual es igual al valor del punto original.

Luego, se escanearán todos los puntos de interés adyacentes a la ubicación actual. Esto se hace mediante un bucle que itera sobre todos los índices de la lista de puntos de interés. Para cada índice, se comprueba si existe una relación entre la ubicación actual y la ubicación representada por este índice en la matriz de adyacencia. Si el valor en la posición [nuevo lugar] [i] en la matriz no es cero, esto indica una conexión entre las dos ubicaciones.

Además de verificar la conexión, también se verifica si la ubicación adyacente no ha sido visitada antes. Esto se hace comprobando el valor correspondiente en la matriz a la que se accede. Si un espacio adyacente cumple con ambas condiciones, se marca como visitado estableciendo el valor en verdadero en la matriz visitada y agregándolo a la cola mediante el método add.

El proceso de extraer un lugar de interés de la cola, escanear sus lugares adyacentes y marcar los lugares adyacentes como visitados, se repite hasta que no haya más entradas en la cola. En otras palabras, el algoritmo continúa descubriendo todos los lugares de interés a los que se puede llegar desde la ubicación original en orden de ancho, es decir, primero los lugares directamente adyacentes, luego los lugares de interés, los puntos adyacentes a las ubicaciones adyacentes.

Cuando se complete el escaneo, se imprimirá una línea en blanco para separar la salida.

En pocas palabras, el algoritmo de búsqueda de ancho (BFS) se utiliza para descubrir un dominio representado por una matriz de adyacencia. Empezando desde un punto de interés inicial y accediendo a todos los puntos de interés accesibles en la misma área utilizando una estructura de cola. Mientras explora, se muestra información sobre los lugares de interés visitados. El algoritmo garantiza que se acceda a las ubicaciones directamente adyacentes antes de cambiar a las ubicaciones adyacentes. Esto garantiza un recorrido por todo el reino.

MST

La función detectHabitualPaths representa el punto de entrada del algoritmo MST (Minimum Spanning Tree). En este punto, se inician las matrices necesarias para construir y almacenar el MST: visited, parent y distance.

La matriz visited es una matriz booleana que realiza un seguimiento de los nodos visitados durante la ejecución del algoritmo. Al principio, todos los elementos se establecen en falso, indicando que ningún nodo ha sido visitado.

La matriz parent es una matriz de enteros que almacena el nodo padre correspondiente a cada nodo en el MST. El nodo raíz se establece en -1.

La matriz distance es una matriz de números de punto flotante que almacena la distancia entre el nodo de origen y cada nodo. Todos los elementos se inicializan con un valor infinito positivo (Float.POSITIVE_INFINITY), excepto el nodo de origen cuya distancia se establece en cero.

El algoritmo utiliza un bucle que itera sobre todos los nodos excepto el último. Esto se debe a que el último nodo ya ha sido agregado al subconjunto de nodos visitados, evitando iteraciones innecesarias. En cada iteración, se utiliza la función findMinimumDistance para encontrar el nodo con la distancia más pequeña.

La función findMinimumDistance se utiliza para encontrar el nodo con la distancia mínima que aún no ha sido visitado durante la construcción del MST. La función recorre todos los nodos del grafo y compara la distancia de cada nodo con el mínimo actual, actualizando las variables min y minIndex cuando se cumplen ciertas condiciones. Al finalizar el bucle, la función devuelve minIndex, que representa el índice del nodo con la distancia mínima, o -1 si no se encuentra ningún nodo que cumpla con las condiciones.

Una vez construido el MST, se muestra información sobre las rutas habituales. Un bucle recorre los nodos desde el segundo hasta el último, obteniendo la información relevante de la matriz de adyacencia adjMatrix y la lista de Puntos de interés. La información se imprime utilizando el método printf para mostrar el origen, destino y distancia de cada ruta en una tabla.

Finalmente, se muestra una línea en blanco para separar la salida del resto del programa y mejorar la legibilidad del resultado obtenido.

DIJKSTRA

En este algoritmo, se utilizan varias estructuras de datos y pasos clave para encontrar el camino más corto desde un nodo de origen hasta un nodo de destino.

El algoritmo comienza solicitando al usuario que ingrese el ID del lugar de origen y el ID del lugar de destino. Estos valores se almacenan en las variables "originID" y "destinationID", respectivamente. También se utiliza la variable "carryingCoconut" para almacenar si la golondrina llevará un coco.

A continuación, se realiza una búsqueda en la matriz de "interestPoints" para encontrar las posiciones de los lugares de origen y destino. Esto se hace mediante un bucle que itera sobre los elementos de la matriz y verifica si el ID del lugar coincide con los valores ingresados por el usuario. Las posiciones de los lugares se almacenan en las variables "originMatrixPosition" y "destinationMatrixPosition".

Después de verificar que los lugares de origen y destino existen en el sistema, se realizan algunas comprobaciones adicionales. Por ejemplo, se verifica si el lugar de origen y el lugar de destino son el mismo. Si es así, se muestra un mensaje de error y se interrumpe la ejecución del algoritmo.

El código también realiza verificaciones basadas en el clima de los lugares de origen y destino. Para ello, se utilizan las variables "onlyEuropeanCanDoIt" y "onlyAfricanCanDoIt". Estas variables almacenan el resultado de comparar el clima de los lugares utilizando el método "equals". Si se cumplen ciertas condiciones climáticas, se realizan cálculos y se muestran resultados específicos.

La función "calculateShortestPath" se utiliza para calcular las rutas más cortas desde el lugar de origen hasta el lugar de destino. Esta función recibe varios parámetros, incluyendo las posiciones de origen y destino, si la golondrina lleva un coco y el tipo de ruta a calcular (europea o africana). Los resultados se almacenan en las listas "pathEuropea" y "pathAfricana".

La función "calculateTimeAndDistance" se utiliza para calcular el tiempo y la distancia de una ruta determinada. Esta función recibe como parámetro la lista de nodos que conforman la ruta y un indicador para determinar si la ruta es europea o africana. Los resultados se devuelven en forma de cadena y se almacenan en las variables "tempsAfricana", "distanciaAfricana", "tempsEuropea" y "distanciaEuropea".

Finalmente, se muestra al usuario la opción más eficiente (europea o africana) junto con el tiempo y la distancia correspondientes. La función "showWinningPath" se utiliza para mostrar el camino ganador, iterando sobre los elementos de la lista de nodos y mostrando información detallada como el nombre del lugar, su ID, la distancia recorrida y el clima asociado.

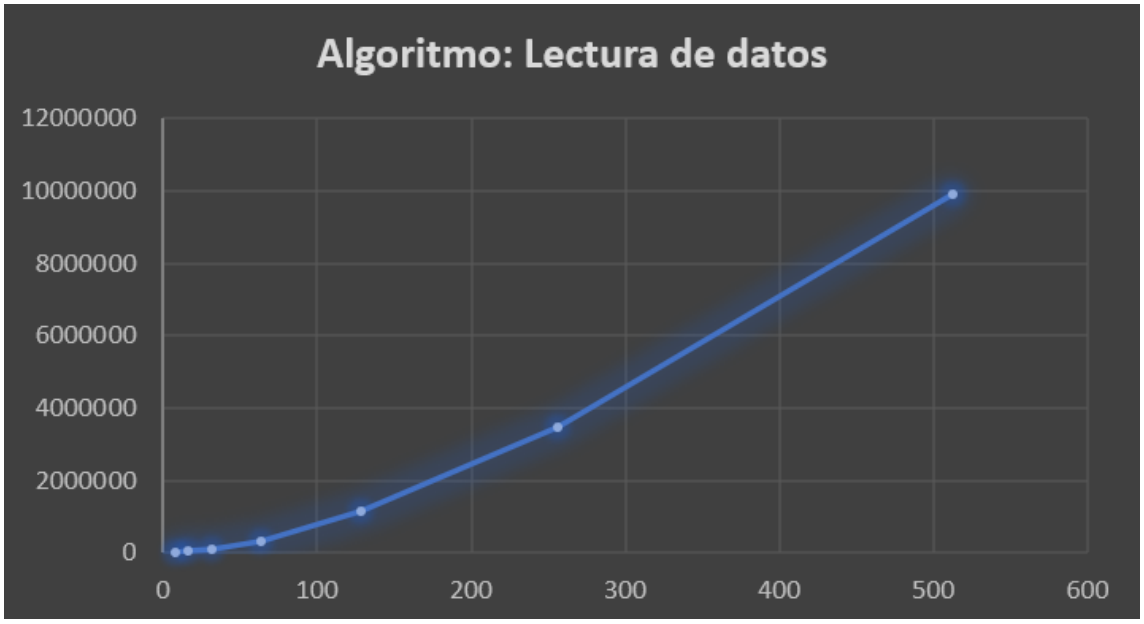
2.1.4. Análisis de rendimiento y resultados

En este apartado, se presentan gráficamente los resultados obtenidos de cada uno de los algoritmos utilizados. Para recopilar la información necesaria, hemos hecho uso de la función de alto rendimiento del sistema Java Virtual Machine, que proporciona el tiempo actual en nanosegundos.

Hemos almacenado el valor devuelto por esta función en una variable de tipo 'long' al principio y al final de la ejecución de cada algoritmo. Al restar estos dos valores, hemos obtenido el tiempo de ejecución del algoritmo en nanosegundos.

2.1.4.1. Algoritmo Lectura de datos

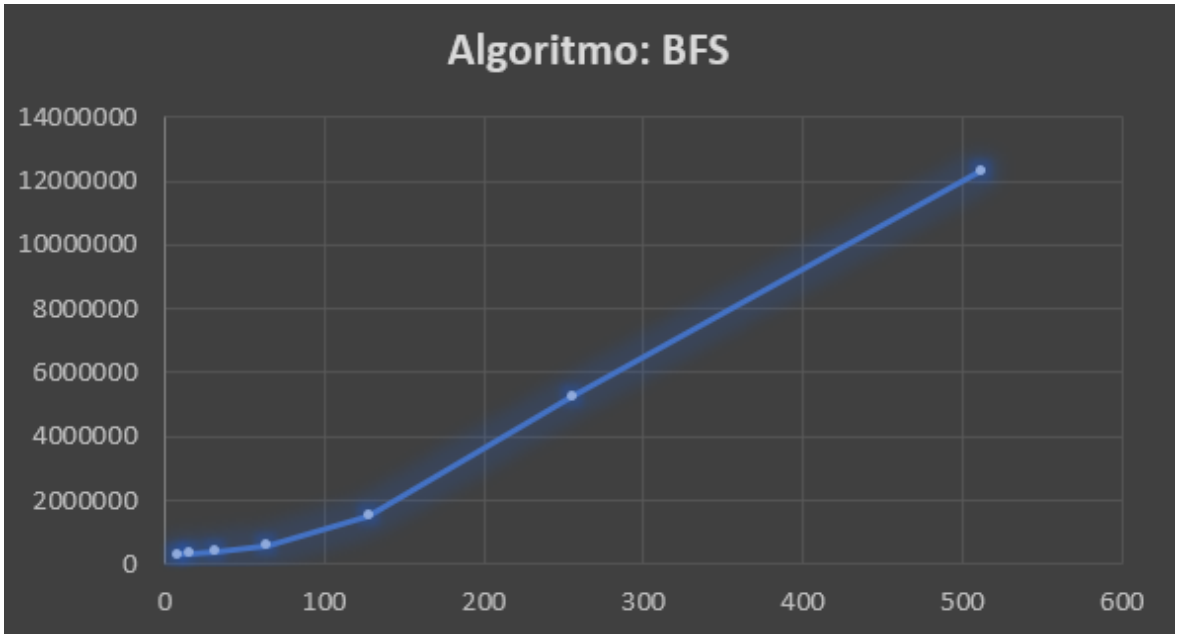
	XXS	XS	S	M	L	XL	XXL
TAMAÑO	8	16	32	64	128	256	512
PRUEBA1	428800	3032900	12449100	24701300	448686600	4974602400	1,77041E+11
PRUEBA2	373600	3040600	3695000	62473100	406358600	8206219400	1,75225E+11
PRUEBA3	401500	2997700	7029400	36100500	408452500	3065218500	1,95511E+11
MEDIA	401300	3023733,33	7724500	41091633,3	421165900	5415346767	1,82593E+11



En el caso de la lectura de datos el coste teorico es proporcional al tamaño del fichero, es decir coste 0(n) donde n es el tamaño del fichero, pero hablando del coste practico puede acercarse al coste teorico, aunque dependiendo de varios factores este se puede no parecerse tanto, entre ellos la velocidad del disco duro, el tamaño del buffer empleado para leer los datos o la carga del sistema en el momento.

2.1.4.2. Algoritmo BFS

	XXS	XS	S	M	L	XL	XXL
TAMAÑO	8	16	32	64	128	256	512
PRUEBA1	425100	360500	387100	617000	1435600	5742400	20118700
PRUEBA2	315200	315200	375800	663100	1614600	5558800	8019500
PRUEBA3	189600	373000	446100	497400	1503300	4544500	8881600
MEDIA	309966,667	349566,667	403000	592500	1517833,33	5281900	12339933,3

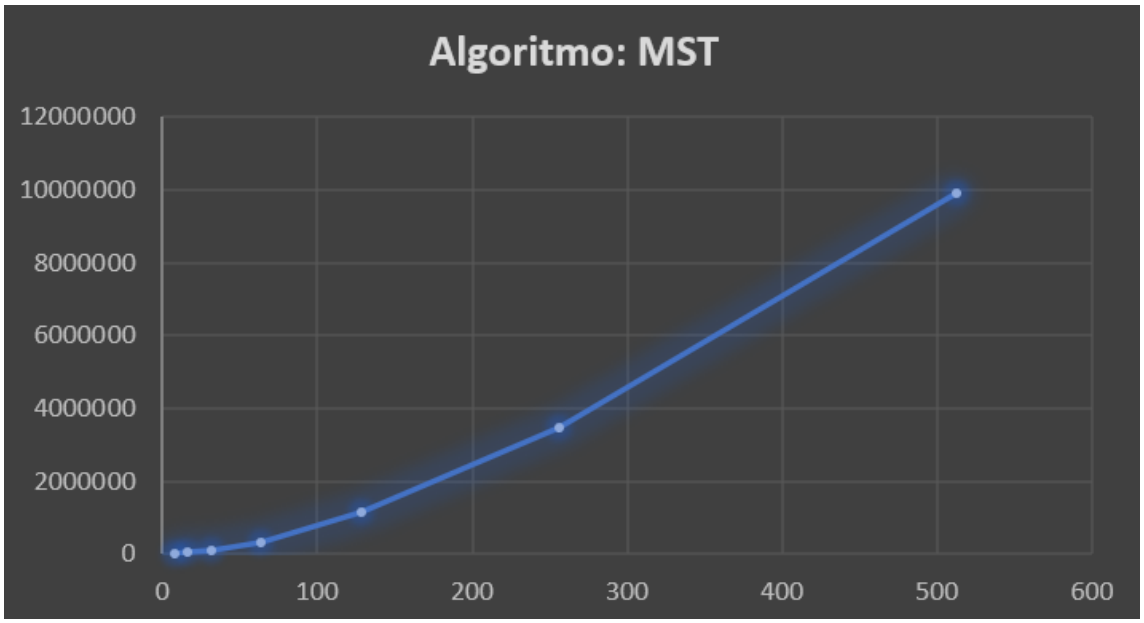


El coste teórico del algoritmo BFS en el grafo con 'V' vertices y 'A' aristas es O(V+A) porque en cada iteracion pasa por un vertice y una arista, recorriendo todo el grafo en horizontal, en el peor

de los casos el coste seria O(V) si se tuvieran que recorrer todos los vertices para llegar a la solucion. En cuanto al coste practico vuelve a depender de varios factores como pasaba en el caso anterior, pero generalmente tiende a acercarse al coste teorico O(V+A)

2.1.4.3. Algoritmo MST

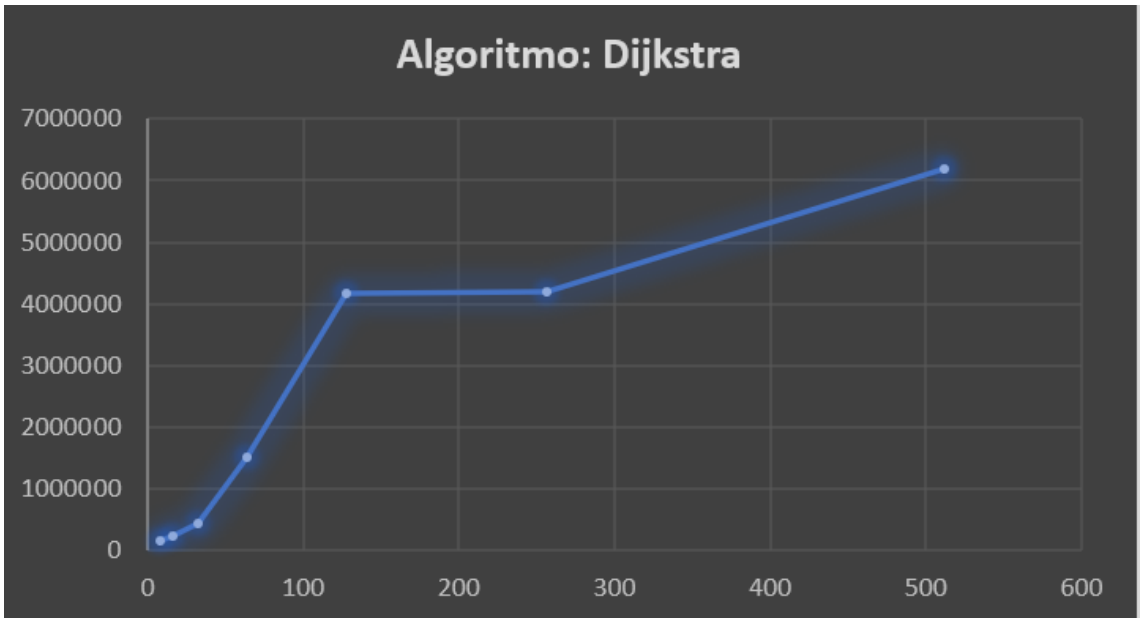
	XXS	XS	S	M	L	XL	XXL
TAMAÑO	8	16	32	64	128	256	512
PRUEBA1	8600	20600	77800	455300	1156100	3659400	8777800
PRUEBA2	10300	41500	93200	213700	1100500	5056100	9194900
PRUEBA3	7100	49600	126000	329700	1238500	1723600	8515700
MEDIA	8666,66667	37233,3333	99000	332900	1165033,33	3479700	8829466,67



En nuestro caso hemos utilizado una matriz de adyacencias para emplear el algoritmo de Prim, por lo que el coste de O(Vertices^2), en el mejor caso el coste sería O(V) ya que se formaría el árbol de expansión mínima.

2.1.4.4. Algoritmo Dijkstra

Columna1	XXS	XS	S	M	L	XL	XXL
TAMAÑO	8	16	32	64	128	256	512
PRUEBA1	172700	368900	444800	1486900	5264100	6235900	7946000
PRUEBA2	125000	145000	352900	1477200	4655400	3833900	6304400
PRUEBA3	160300	87400	565500	1239500	2732100	3008300	6239800
PRUEBA4	103300	115400	475000	1277400	3183500	3085800	5642300
PRUEBA5	182100	201100	400100	1720300	4095800	5071800	6029400
PRUEBA6	215300	546100	384000	1849200	5062200	4023900	5018200
MEDIA	159783,333	243983,333	437050	1508416,67	4165516,67	4209933,33	6196683,333



Para el algoritmo dijkstra el coste teórico seria O((V+A)log V), como utiliza una cola de prioridad en su implementación, lo que obliga a tener un coste de log V por cada inserción y extracción de la cola. En el coste práctico puede acercarse bastante al teórico en general. En el peor caso seguiría manteniéndose como O((V+A)log V) pero en el mejor caso solo habría una ruta a cada vértice, en cual caso sería O(V log V).

2.1.5. Método de pruebas utilizado

En esta sección, hemos aprovechado los recursos de prueba y depuración disponibles en IntelliJ IDEA, un entorno de desarrollo integrado (IDE) altamente eficaz. Estos instrumentos han sido fundamentales para mejorar tanto la calidad como el rendimiento de nuestro software.

La aplicación de estas herramientas de prueba y depuración en IntelliJ IDEA nos ha permitido inspeccionar a profundidad el flujo de ejecución de nuestro código, identificando posibles errores o fallos. Establecimos puntos de interrupción en el código que nos permitieron pausar la ejecución y examinar el estado de las variables y objetos en tiempo real. Esto ha resultado especialmente útil para detectar errores lógicos, seguir la traza de los datos y solucionar posibles problemas de rendimiento.

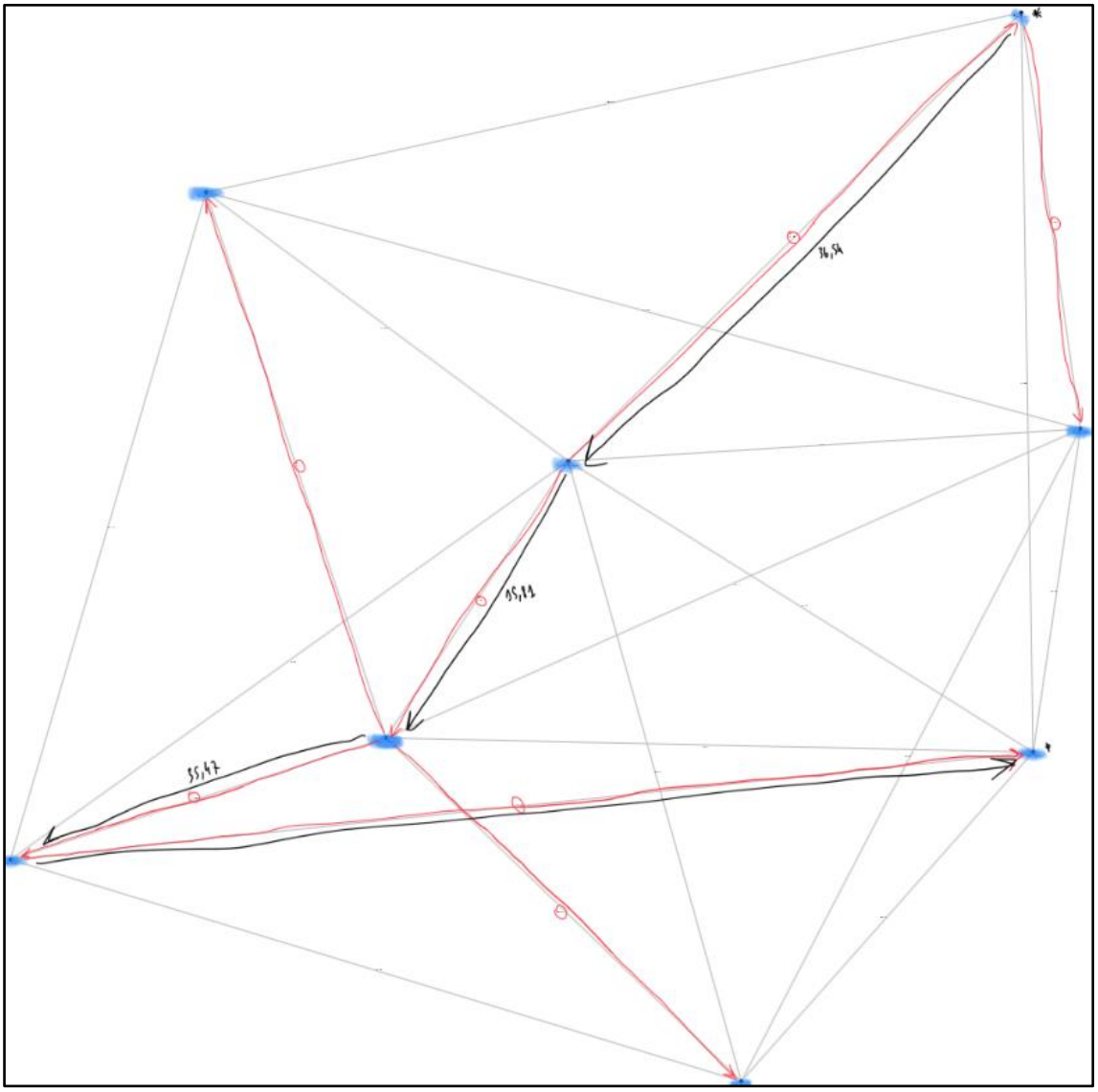
Además, hemos utilizado la función de depuración y el 'debugger' de IntelliJ IDEA junto con el uso de impresiones en consola, lo cual nos ha facilitado una visión más detallada de cómo se está ejecutando y comportando nuestro código.

Paralelamente, IntelliJ IDEA proporciona una gama diversa de herramientas de prueba, incluyendo pruebas unitarias, de integración y de regresión. Estas nos han permitido comprobar la funcionalidad de los distintos componentes de nuestro software, asegurando que cumplen los requisitos y se comportan adecuadamente bajo distintas circunstancias.

En un enfoque más manual, utilizamos un iPad y el Apple Pencil para trazar el camino y verificar que era correcto directamente en los archivos. Esta técnica nos permitió visualizar el flujo del programa de manera intuitiva y detectar rápidamente cualquier anomalía.

El empleo de estos métodos de prueba y depuración en IntelliJ IDEA ha incrementado la calidad de nuestro software y ha acelerado nuestro proceso de desarrollo. Hemos logrado identificar y corregir errores de manera más eficaz, ahorrando tiempo y esfuerzo a largo plazo. Además, al

detectar y resolver problemas en etapas tempranas del desarrollo, hemos conseguido ofrecer un producto final de mayor calidad.



Comprovación manual de los algoritmos mediante Ipad & ApplePencil

2.1.6. Problemas observados

Durante la implementación de los algoritmos BFS, MST y Dijkstra en nuestro proyecto, nos enfrentamos a varios desafíos. En el caso del algoritmo BFS, tuvimos un problema de manejo incorrecto de los nodos visitados, lo que causaba un bucle infinito de procesamiento. Sin embargo, logramos resolver este problema al corregir la marcación de los nodos visitados, permitiendo que el algoritmo explorara correctamente todos los nodos del grafo.

En cuanto al algoritmo MST, nos encontramos con dificultades en la detección y manejo de ciclos en el grafo durante la generación del árbol de expansión mínimo. Nuestra implementación inicial no verificaba adecuadamente la existencia de ciclos, lo que resultaba en la creación de un MST con ciclos. Para solucionar esto, incorporamos una estrategia para evitar la inclusión de aristas que generaran ciclos en el árbol, lo que nos permitió obtener un MST válido sin ciclos.

En relación al algoritmo Dijkstra, nos enfrentamos a problemas de rendimiento en grafos de gran tamaño. A medida que aumentaba el número de nodos y aristas, el algoritmo se volvía considerablemente más lento y consumía una cantidad significativa de recursos. Para abordar esto, implementamos optimizaciones como el uso de estructuras de datos más eficientes, como montículos binarios, y la aplicación de técnicas de poda y mejora de la eficiencia en la búsqueda de rutas óptimas. Estas optimizaciones nos permitieron mejorar el rendimiento del algoritmo Dijkstra, reduciendo el tiempo de ejecución y el consumo de recursos en grafos grandes. Aprendimos la importancia de optimizar y ajustar nuestros algoritmos para hacer frente a diferentes desafíos durante la implementación.

2.2. Árboles de búsqueda binarios auto-balanceados (AVL Trees)

2.2.1. Introducción

Los árboles AVL son una variante de los árboles de búsqueda binarios que garantizan un balance óptimo en su estructura. A través de la implementación de rotaciones, los árboles AVL mantienen una altura equilibrada, lo que asegura que las operaciones de búsqueda, inserción y eliminación se realicen en tiempo logarítmico. Esta característica hace que los árboles AVL sean una opción eficiente para almacenar y recuperar información ordenada en aplicaciones donde se requiere un alto rendimiento en operaciones de búsqueda y actualización.

2.2.2. Diseño

En nuestra implementación hemos representado los árboles a partir de un nodo raíz. Teniendo en cuenta eso, podemos crear una clase nodo que guarde dos posibles hijos: izquierda y derecha. Un nodo también ha de poder ser una hoja, por lo tanto, tendrá los dos hijos igualados a “null”.

Para realizar la optimización y auto balance de este tipo de árboles también tenemos que guardar un factor de balanceo con el cual saber que opción usar. El factor de balanceo debe estar comprendido entre [-1, 1] para que se considere un árbol óptimo, en caso contrario, deberemos revalanzearlo.

Como los hijos de los nodos pueden ser null, debemos tener en cuenta que orden tienen. Si son:

- Orden 0: Dos hijos igual a null
- Orden 1: Un hijo igual a null
- Orden 2: Los dos hijos apuntan a otros nodos o hojas

En función de esto podemos crear el factor de balanceo previamente descrito que relacionará cada nodo con la raíz.

2.2.3. Algoritmos implementados

Para esta estructura de datos no solamente nos piden implementarla, sino que también tenemos que crear diferentes algoritmos para usarla.

2.2.3.1 Insertar Habitante

Para añadir un habitante primero comparamos el valor a insertar (Id) con el valor del nodo actual. Si es menor, se repite el proceso en el subárbol izquierdo; si es mayor, se repite en el subárbol derecho. Este proceso continúa hasta encontrar una posición adecuada para el nuevo valor.

Una vez encontrado, creamos un nuevo nodo y se enlaza correctamente con el árbol existente. Al finalizar, el árbol binario mantiene su estructura ordenada y balanceada, permitiendo búsquedas eficientes y otras operaciones relacionadas.

2.2.3.2 Eliminar Habitante

A la hora de eliminar un habitante, comenzamos por buscar el nodo que queremos eliminar (empezando por la raíz y bajando por la izquierda o derecha dependiendo del valor que queremos encontrar).

Una vez localizamos en el nodo, hay que tener en cuenta varias situaciones. Si el nodo es una hoja, simplemente se elimina. Si el nodo tiene dos hijos, buscamos el sucesor inmediato y lo remplazamos por el nodo a eliminar. Si el nodo solo tiene un hijo, remplazamos directamente el nodo a eliminar por su sucesor.

Hay que tener en cuenta que, una vez realizada la eliminación, siempre balanceamos el árbol para que quede una estructura ordenada.

2.2.3.3 Identificar Brujas

Para hacer una búsqueda por el árbol binario comparamos el valor a buscar con el valor inicial del árbol (la raíz) y dependiendo del resultado de esa comparación continuamos por el sucesor izquierdo o derecho.

Hacemos esto de manera recursiva hasta que encontramos el valor a buscar dentro del nodo.

2.2.3.4 Batuda

En este apartado buscamos dentro de un rango determinado en el arbol binario. Para hacer esto, comenzamos comparando el rango con el valor del nodo actual. Si el valor del nodo se encuentra dentro del rango, se agrega a la lista de resultados.

Luego, se realiza una búsqueda recursiva en los subárboles izquierdo y derecho, considerando únicamente aquellos subárboles que potencialmente podrían contener valores dentro del rango. (Por ejemplo, descartando el subárbol izquierdo si el valor actual es más pequeño que el rango mínimo)

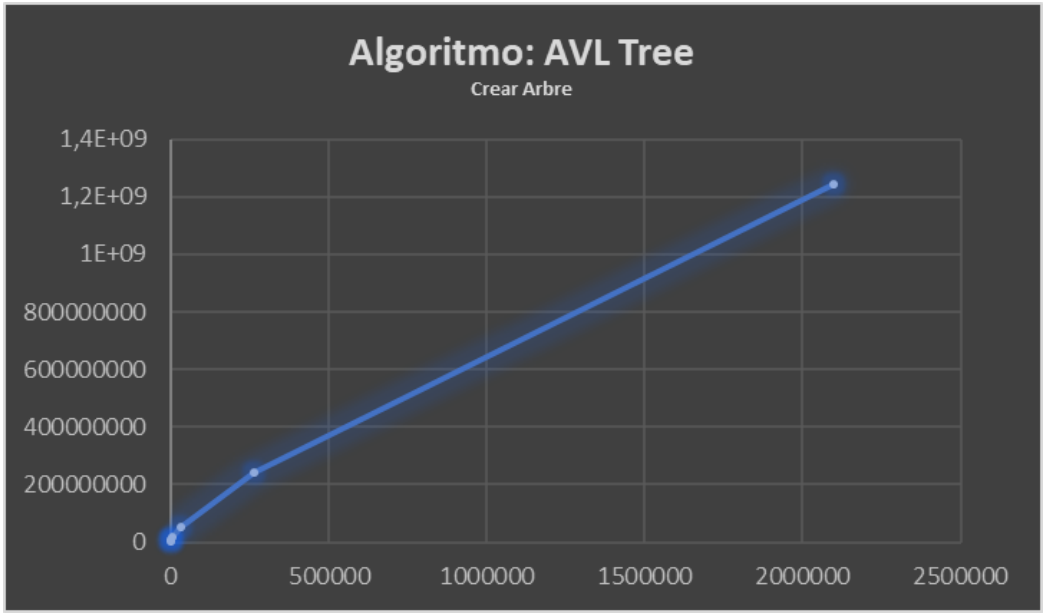
2.2.4. Análisis de rendimiento y resultados

Para probar el tiempo de estas funciónes hemos usado la función “System.nanoTime()” que devuelve en una variable long, el tiempo en nanosegundos actuales.

Si usamos esta función antes y después del algoritmo, podemos hacer la diferencia para saber cuánto tiempo ha tardado dicho algoritmo en ejecutarse.

2.2.4.0. Crear Arbol

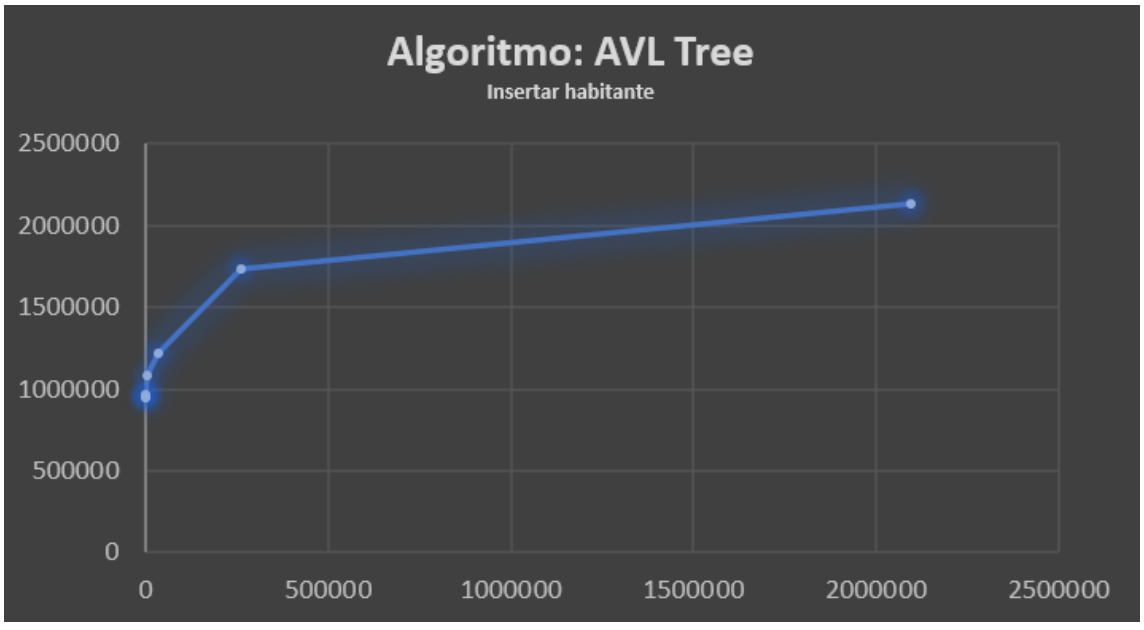
	XXS	XS	S	M	L	XL	XXL
Test 1	11191200	10883100	18417700	34303300	90668700	439684100	3736172500
Test 2	10527900	1465200	7177800	14435300	60211900	286918700	3474967600
Test 3	593300	520700	736000	2903600	22281000	250618100	3516147600



El costo total de inserción será la suma de los costos individuales de inserción de cada valor. Por lo tanto, ja que en el caso promedio el coste es O(log n), el costo total de inserción de "m" valores sería aproximadamente O(m log n), mientras que en el peor caso sería O(mn), donde "m" es el número de valores a insertar.

2.2.4.1. Afegir habitant

Columna1	XXS	XS	S	M	L	XL	XXL
TAMAÑO	8	16	32	64	128	256	512
PRUEBA1	945300	958900	964800	1086900	1236900	1736900	2136900
PRUEBA2	941300	945600	957600	1069800	1126400	1543500	2027900
PRUEBA3	962300	972300	978900	1093200	1289800	1934300	2238400

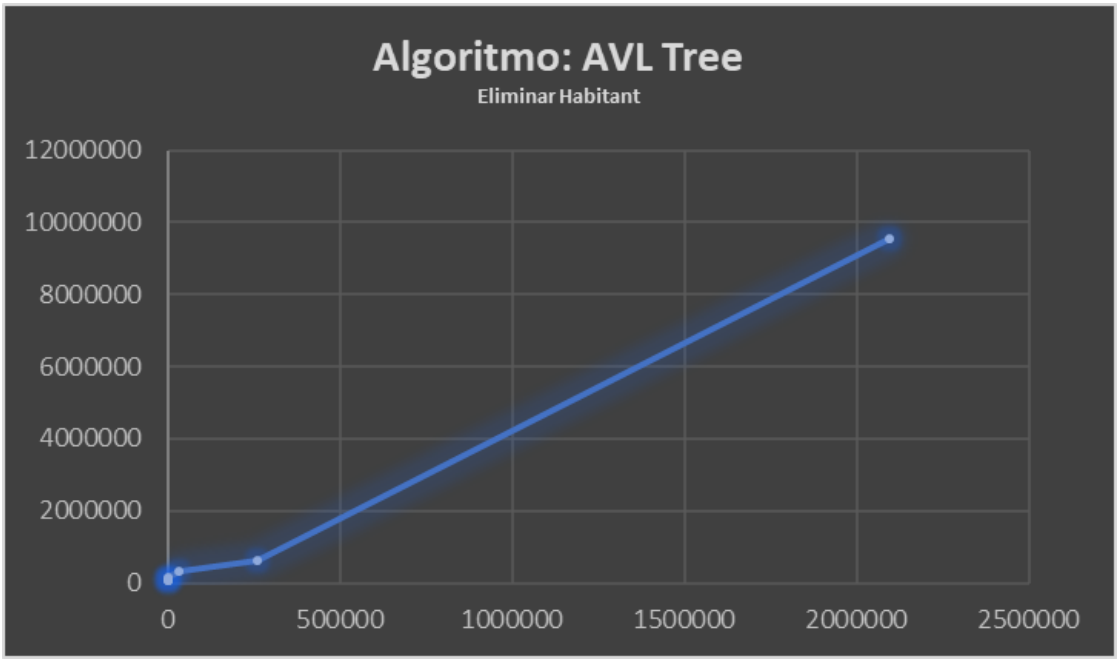


En el peor caso, cuando el árbol está desequilibrado y tiene una estructura lineal, la inserción puede tener un costo de O(n), donde "n" es el número de nodos en el árbol.

Sin embargo, en el caso promedio, cuando el árbol está equilibrado, la inserción tiene un costo de O(log n), ya que se realiza una búsqueda binaria para encontrar la posición adecuada para insertar el nuevo valor.

2.2.4.2. Eliminar habitant

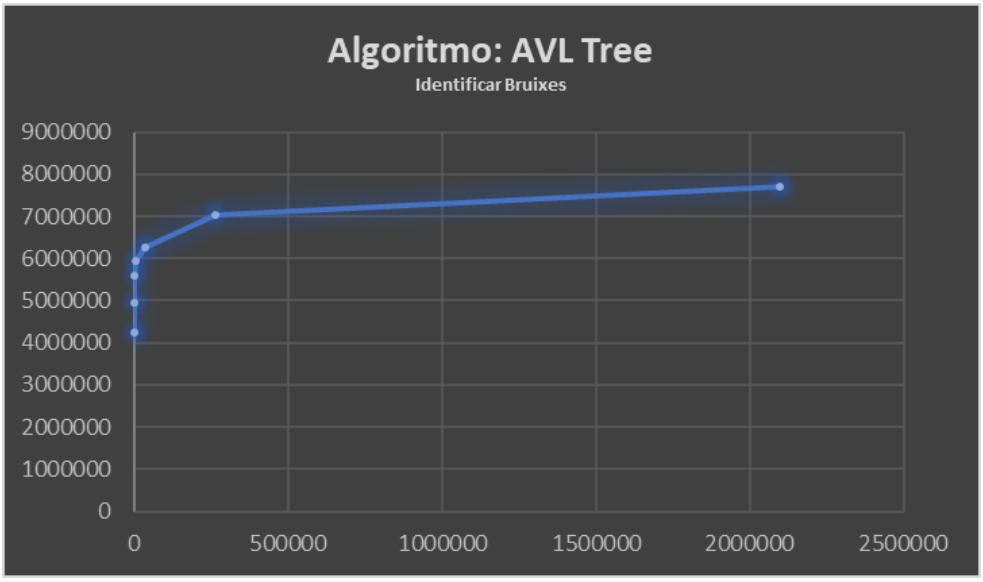
Columna1	XXS	XS	S	M	L	XL	XXL
TAMAÑO	8	16	32	64	128	256	512
PRUEBA1	53000	80300	123000	153400	307200	632900	9534300
PRUEBA2	54300	83400	135300	163400	314200	664400	9376500
PRUEBA3	51400	87500	111200	145700	376400	623300	9644500



Al igual que la inserción, la eliminación puede tener un costo de O(log n) si el árbol está equilibrado. Sin embargo, en nuestro caso, la eliminación tiene un costo de O(n) debido a que en lugar de hacer la búsqueda binaria para encontrar el valor a eliminar recorremos todo el arbol hasta encontrar el valor.

2.2.4.3. Identificació de bruixes

	XXS	XS	S	M	L	XL	XXL
Test 1	10293400	12343300	14204100	15137900	15363200	16892500	18323400
Test 2	9545300	11881600	14257900	15683600	15945600	17769400	19359300
Test 3	9357600	11221100	12961700	15457800	15496000	16154400	17599300

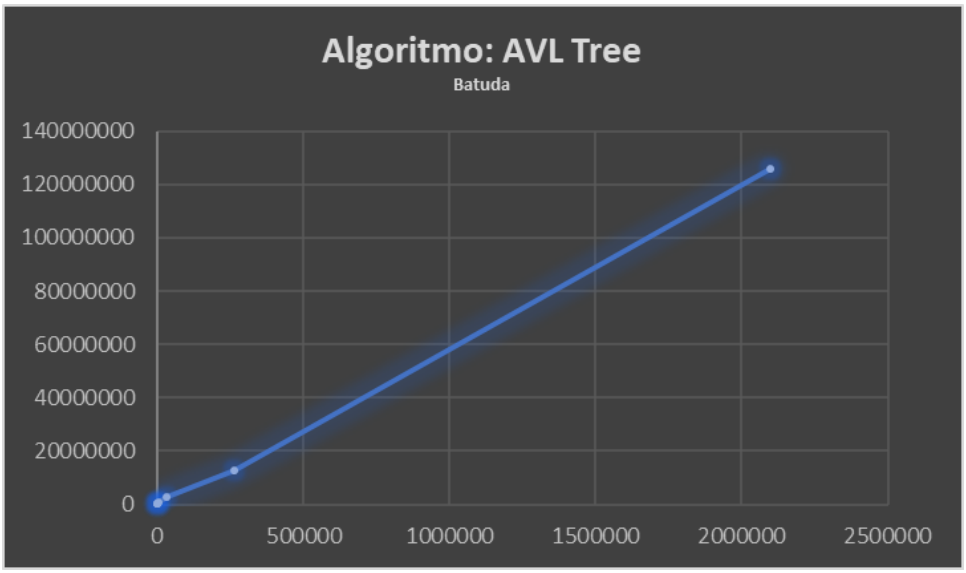


Viendo la gráfica podemos asegurar que la búsqueda puede tener un costo de O(n) en el peor caso, ya que se debe recorrer todo el árbol para encontrar el valor deseado. Sin embargo, en un árbol binario equilibrado, la búsqueda tiene un costo de O(log n) en el peor caso y en el caso promedio.

Comparando el coste teórico con la gráfica que hemos obtenido, nuestras conclusiones se confirman ya que la trayectoria es logarítmica.

2.2.4.4. Batuda

	XXS	XS	S	M	L	XL	XXL
Test 1	16802000	17370000	18236400	18534700	19635400	21374500	26431300
Test 2	16453400	17123400	18233500	18234400	20231200	23234300	29834300
Test 3	16234400	17532300	18123200	18823200	22321200	26546500	27361200



Como podemos observar el coste de esta función sería O(k + log n), donde "k" es el número de valores dentro del rango y "n" es el número de nodos en el árbol. En la mayoría de los casos, el costo se acerca a O(k) si el rango abarca una fracción significativa de los valores totales en el árbol.

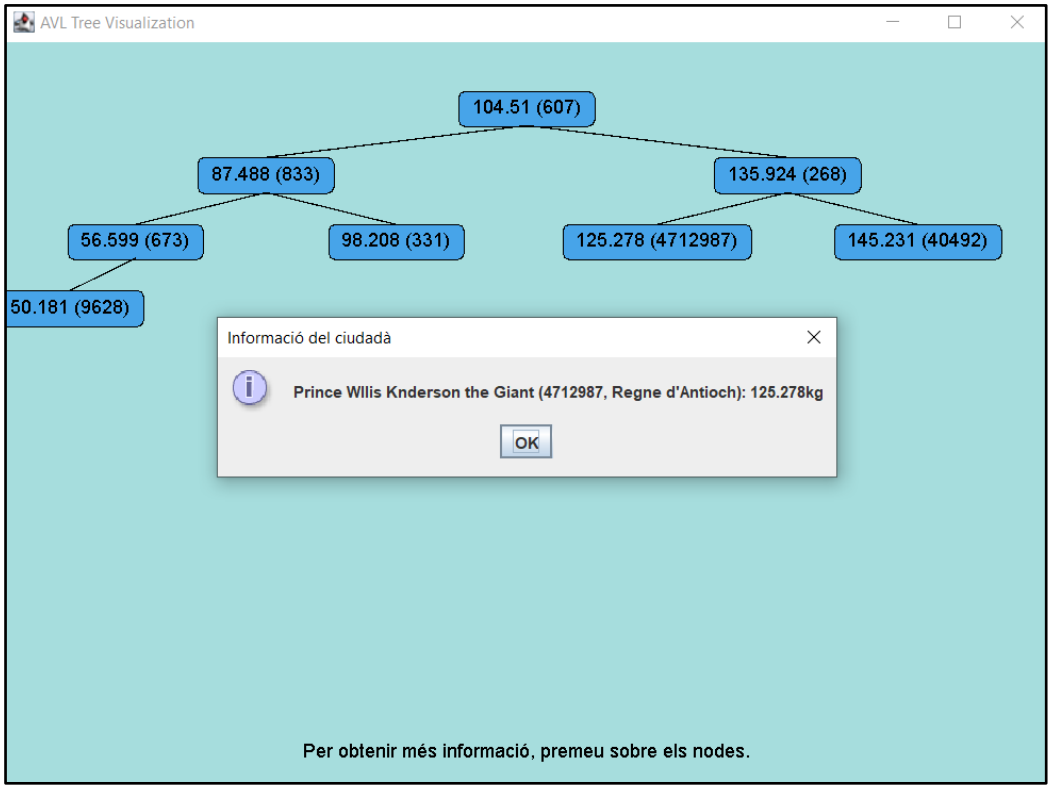
Debido al coste, cuanto mayor sea el repertorio de datos disponibles mayor será el tiempo de ejecución y escalará de forma extrema en el peor de los casos recorriendo así todo el árbol.

2.2.5. Método de pruebas utilizado

Se ha utilizado un JFrame para representar visualmente el árbol binario y permitir la interacción con sus nodos. El JFrame es una ventana gráfica en Java que proporciona una interfaz de usuario interactiva. Mediante el uso de componentes gráficos y eventos, se ha implementado la lógica para mostrar los nodos del árbol y permitir al usuario hacer clic en ellos.

Cada nodo del árbol se ha representado visualmente como un objeto gráfico dentro del JFrame. Al hacer clic en un nodo, se ha configurado un evento que captura la interacción del usuario. Al detectar el clic, se ha diseñado la lógica correspondiente para mostrar el valor asociado con el nodo seleccionado.

Esto puede implicar la visualización de una ventana emergente con la información del nodo o mostrar el valor directamente en una ubicación designada dentro del JFrame. Esta interacción proporciona una forma conveniente y visualmente atractiva para observar los valores de los nodos en el árbol binario.



2.2.6. Problemas observados

Tuvimos problemas a la hora de representar los árboles ya que los outputs de la consola se veían muy diferentes a los outputs teóricos que habíamos predicho. Conseguimos solucionarlo y hasta implementamos una interfaz gráfica para visualizarlos mejor.

2.3. Árboles balanceados por altura (R Trees)

2.3.1. Introducción

Los R-Trees son estructuras de datos jerárquicas y multidimensionales diseñadas para indexar y recuperar información geoespacial y objetos espaciales en bases de datos. Estos árboles se utilizan en aplicaciones de Sistemas de Información Geográfica (GIS), búsqueda de vecinos más

cercanos y reconocimiento de patrones. La principal ventaja de los R-Trees radica en su capacidad para manejar eficientemente grandes cantidades de datos espaciales y realizar consultas de rango rápidas en múltiples dimensiones.

2.3.2. Diseño

Al diseñar esta estructura de datos no lineal, hemos aplicado conceptos avanzados y sofisticados de ciencias de la computación y geometría computacional, optimizando el rendimiento y la eficiencia en la manipulación de entidades geométricas en un espacio bidimensional.

En primer lugar, la clase RTree actúa como un contenedor que alberga el nodo raíz, el cual a su vez contiene otros nodos o entidades geométricas denominadas Bardizas. Cada nodo en la estructura del árbol R representa un rectángulo mínimo delimitador (Minimum Bounding Rectangle, MBR) que engloba todos los rectángulos o elementos geométricos contenidos en él.

La clase Node se define por atributos clave, como bardizas, rectangles, children, bounds y parent, que permiten la representación jerárquica de la estructura y facilitan la implementación de algoritmos de búsqueda y modificación.

Los nodos hoja almacenan objetos Bardiza que representan entidades geométricas con atributos específicos, como tipo, tamaño, latitud, longitud y color. En cambio, los nodos no hoja albergan descendientes rectangulares en forma de nodos hijos, los cuales representan MBR que engloban otros nodos o elementos geométricos. Esta separación de responsabilidades permite una mayor eficiencia en la manipulación de la estructura del árbol.

Para garantizar la integridad y el rendimiento de la estructura de datos, hemos desarrollado algoritmos de inserción (insertNode()), eliminación (deleteNode()) y búsqueda (searchNode()) que garantizan el equilibrio y la optimización del árbol R. Estos algoritmos emplean heurísticas y estrategias de división y ajuste, como la técnica de dividir y conquistar (splitNode()) y el método de agrupamiento de rectángulos mínimos delimitadores, que se lleva a cabo mediante el proceso de (chooseLeaf()) y (adjustTree()).

Además, hemos creado clases auxiliares, como Bardiza y BardizaDistance, que permiten el manejo eficiente de entidades geométricas y la computación de distancias entre objetos en el espacio bidimensional.

Para mejorar aún más la calidad del código y la estructura de datos, se podría considerar la aplicación de principios de diseño orientado a objetos, como la herencia y la abstracción. Por ejemplo, la clase Node podría heredar de una clase geométrica abstracta que defina las operaciones básicas y necesarias empleadas en los algoritmos implementados. Esto permitiría un mayor modularidad y una mejor separación de responsabilidades, lo que redundaría en una mayor legibilidad y mantenibilidad del código.

2.3.3. Algoritmos implementados

2.3.3.1. Añadir Bardiza

La inserción en un R-Tree se realiza de la siguiente manera:

- Se selecciona una hoja para la inserción usando el método “chooseLeaf()”. Este método empieza en la raíz y selecciona el hijo que requiere el menor aumento en su área de rectángulo delimitador mínimo (MBR) para acomodar el nuevo objeto. Si hay un empate, se elige el rectángulo con la menor área.

- Una vez seleccionada la hoja, se añade el nuevo objeto. Si la hoja está sobrellena (es decir, contiene más entradas de las permitidas, en nuestra implementación le hemos puesta máximo 4, pero se puede modificar a gusto del usuario), se divide usando el método “splitNode()”. El proceso de división selecciona dos objetos para ser las primeras entradas de los dos nuevos nodos, y luego asigna los objetos restantes al nodo que requiere el menor aumento en el área del MBR.

- Después de la división, el método adjustTree se encarga de propagar los cambios hacia arriba en el árbol, actualizando los MBRs y dividiendo los nodos si es necesario con la función “splitNode()”. Si la raíz es dividida, se crea una nueva raíz.

2.3.3.2. Eliminar Bardiza

La eliminación de una Bardiza comienza buscando la hoja que contiene el objeto, luego se elimina el objeto y se actualizan los MBRs en el camino de vuelta a la raíz utilizando el método “updateMBRsAfterDelete()”.

2.3.3.3. Visualización

Para la visualización del árbol nos centraremos en dos funciones clave que facilitan esta visualización:

- **showRTreeInVertical():** Esta función imprime la estructura del R-tree en un formato vertical.
  - Primero, crea un objeto StringBuilder para construir la representación de cadena del árbol.
  - Si el nodo raíz del árbol es nulo, simplemente agrega "Empty tree" a la cadena.
  - Si el nodo raíz no es nulo, agrega la representación formateada del MBR (Minimum Bounding Rectangle) del nodo raíz a la cadena.
  - Luego, recorre los nodos hijos del nodo raíz. Para cada nodo hijo, verifica si es el último hijo y si su MBR está dentro de un rango específico. Si es así, agrega una representación de cadena del nodo hijo al StringBuilder.
  - Si el nodo hijo no es una hoja, actualiza la lista de nodos hijos para ser los hijos del nodo hijo actual y reinicia el índice del bucle.
  - Si el nodo hijo es una hoja, recorre los objetos "Bardiza" en el nodo hijo y agrega una representación de cadena de cada "Bardiza" al StringBuilder.
  - Finalmente, imprime la representación de cadena del árbol.

- **formatMBR(Rectangle2D.Double mbr)**: Esta es una función auxiliar que toma un objeto MBR y devuelve una representación de cadena de ese MBR. La representación de cadena incluye las coordenadas ”x” e “y” del MBR, así como su ancho y alto.

2.3.3.4. Búsqueda por área

La búsqueda en nuestro R-Tree es muy eficiente debido a su estructura jerárquica. El método “searchByRange()” busca todos los objetos dentro de un rango especificado. Comienza en la raíz y se desplaza descendentemente en el árbol, comprobando si el MBR de cada nodo intersecta con el rango de búsqueda introducido por el usuario. Si es así, se comprueban sus hijos, y así sucesivamente. Si se encuentra un nodo hoja, se comprueban todas sus Bardizas.

2.3.3.5. Encontrar los K vecinos

Este código también implementa una búsqueda del vecino más cercano, que encuentra los k objetos más cercanos a un punto dado. Esto se logra mediante el método

"findKNearestBardizas()", que utiliza una cola de prioridad para mantener los k objetos más cercanos encontrados hasta ahora, y los va actualizando a tiempo real a medida que va explorando el árbol. Posteriormente, como nos piden el color medio, y el tipo majoritário de estas k bardizas, hemos implementado dos funciones, aunque se puede implementar en una sola, pero para mejor legibilidad de codigo, hemos preferido hacerlo en dos funciones separadas:

- "getMajorityType()": la cual mediante un HashMap llamado typeCount cuenta el número de apariciones de cada tipo de bardiza. Se recorre la lista de bardizas más cercanas y actualiza el conteo en typeCount. Finalmente, devuelve el tipo que tiene el conteo más alto utilizando "Collections.max()" y "Map.Entry.comparingByValue()".

- "getMediumColor()": calcula el color promedio de las k bardizas más cercanas a una ubicación (latitude, longitude). Recorre la lista de bardizas como la función anterior (por eso previamente hemos mencionado que se pueden unir en una sola función) y suma los valores de los componentes RGB de cada color. Al dividir las sumas por k, se obtiene el promedio de los componentes rojo, verde y azul, el cual se formatea en formato hexadecimal (#RRGGBB) y se devuelve como una cadena.

2.3.4. Análisis de rendimiento y resultados

Para cuantificar la eficiencia temporal de estas funciones, hemos recurrido a la precisión de la función System.nanoTime(), una herramienta incorporada en el lenguaje de programación Java que proporciona la marca de tiempo actual en nanosegundos, almacenada en una variable de tipo long.

La metodología que hemos empleado para medir el tiempo de ejecución del algoritmo implica la invocación de System.nanoTime() en dos puntos críticos del proceso: inmediatamente antes de que el algoritmo comience su ejecución y justo después de que finalice. La resta de estos dos valores nos proporciona un cálculo preciso de la duración de la ejecución del algoritmo, expresado en nanosegundos.

Este enfoque nos permite realizar un análisis detallado y riguroso del rendimiento temporal de las funciones, proporcionando una métrica cuantitativa de su eficiencia. Esta medida es esencial para la optimización y el ajuste de rendimiento, permitiéndonos identificar y mejorar cualquier cuello de botella potencial en el código.

2.3.4.1. Resultados de insertar todas las bardizas de los ficheros

IMPORTANTE Debido a que nuestra implementación de inserción del RTree es poco óptima, hemos observado problemas en cuanto a tiempo de ejecución para la inserción del árbol, ya que, si nos basamos en el coste teórico que debería tener, O(log(n)), no nos acercamos a él, ya que nuestro coste es exponencial. Es por ello, que hemos tenido que personalizar las entradas de nodos en el árbol, ya que el fichero M nos tardaba 3 horas. Esto ha podido alterar algunas gráficas, debido a que el coste puede ser confuso entre coste O(n) o O(log(n)). Para evitar esto se proporcionaban datasets con números de entrada muy distantes entre ficheros, con la finalidad de poder graficar y ver el coste correctamente. Aun así, hemos investigado por internet el coste teórico que debería tener, y pese a que hemos encontrado poca información al respecto, hemos podido conseguir el coste teórico que se debería lograr para cada función.

El tiempo exponencial del fichero M no lo hemos comprobado esperando, pero si de manera teórica, ya que estuvimos esperando 15 minutos y al ver que no finalizaba, decidimos comprobar que ocurría.

Para calcular de manera teórica el tiempo que tardaban los ficheros hicimos uso de una formula matemática para predecir el tiempo de manera exponencial del algoritmo en función del tamaño del dataset de entrada.

Sabiendo que el tiempo con 2048 bardizas es de 29 segundos y con 4096 de 214 segundos, podemos calcular lo siguiente:

- 29=a*(b^2048)
- 214=a*(b^4096)

Dividiendo los exponentes entre 2048 nos queda lo siguiente:

- 29=a*(b^1)
- 214=a*(b^2)

Y esto es un sistema de ecuaciones de dos ecuaciones con dos incógnitas, cuyas soluciones són: a = 841/214 y b = 214/29.

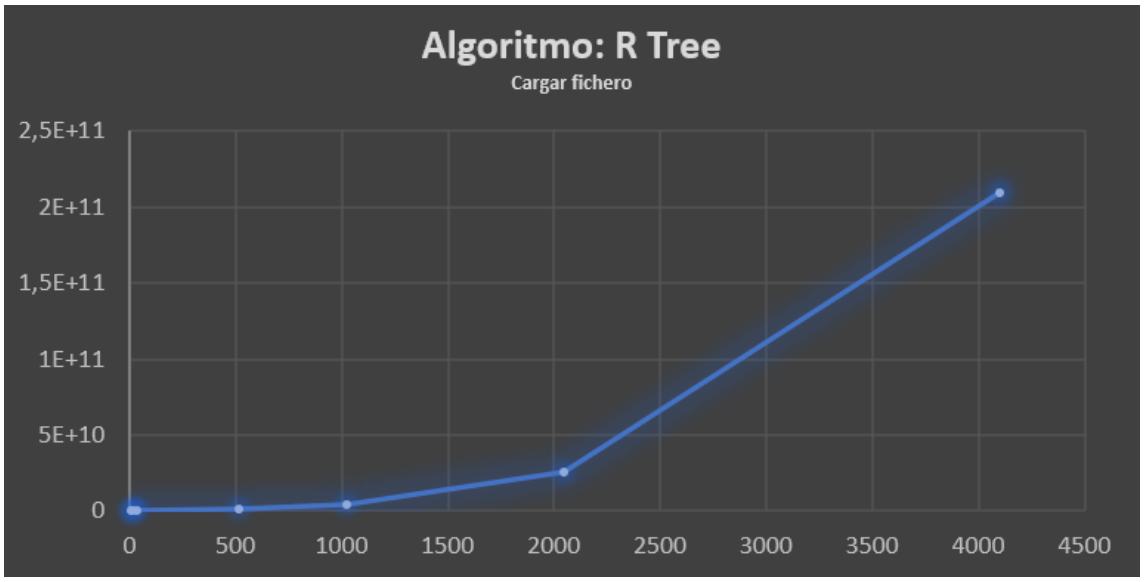
Teniendo los valores de a y b, podemos usarlos para predecir el tiempo de procesamiento por ejemplo del archivo M con 8192 elementos. La ecuación sería:

tiempo = a*(b^(x/(8192/2048))) = 9800344/841 = 11653.2 segundos = **3,23694444** horas

Del mismo modo, si calculamos el tiempo del fichero más grande (XXL) con 524.288 bardizas, 6.403549009026405×10^222 segundos = **2.030552070340692×10^215** años

Es por ello, que para el análisis de la inserción hemos personalizado los números de bardizas de los datasets, llegando a un máximo de 4096. Aquí están los resultados:

	XXS	XS	S	PERSONALIZAD	PERSONALIZAD	PERSONALIZAD	PERSONALIZAD
TAMAÑO	8	16	32	512	1024	2048	4096
PRUEBA1	2112500	2477500	4632800	817141100	3843682900	26011355800	2,12102E+11
PRUEBA2	1924900	2375900	4442600	818745100	3788174800	25637232100	2,04419E+11
PRUEBA3	2036200	2334400	4727600	824803100	3997820300	25605413300	2,11273E+11
MEDIA	2024533,33	2395933,33	4601000	820229766,7	3876559333	25751333733	2,09265E+11



Como se mencionó anteriormente, estamos observando un coste exponencial O(n^m), aunque teóricamente deberíamos esperar un coste O(log(n)) en un R-tree. Este comportamiento puede atribuirse a dos factores principales:

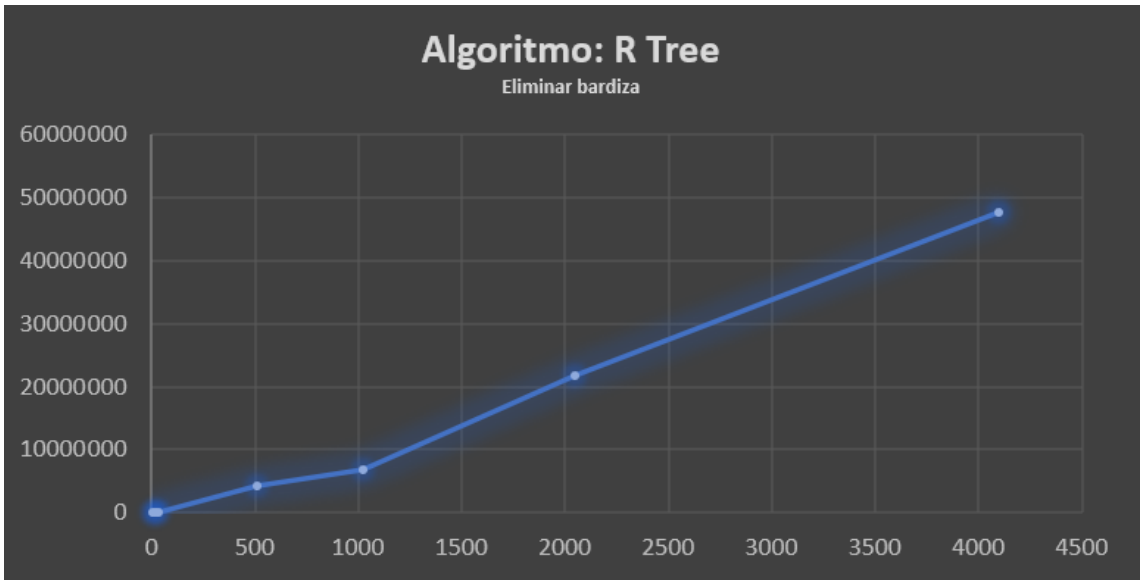
- Primero, la metodología que utilizamos para la creación de los rectángulos es exponencial en su naturaleza. Aunque somos conscientes de esta limitación, no hemos logrado mitigar completamente su impacto en el rendimiento del algoritmo.

- Segundo, la cantidad exponencial de "overflows" que ocurren durante la inserción en el árbol puede contribuir a alejarnos de un coste logarítmico. Estos "overflows" pueden aumentar la profundidad del árbol y, por lo tanto, el número de nodos que deben ser visitados durante una búsqueda.

Reconocemos que estos factores están afectando la eficiencia de nuestro algoritmo y hemos trabajado en soluciones para poder abordar estos problemas.

2.3.4.2. Resultados de eliminar una Bardiza

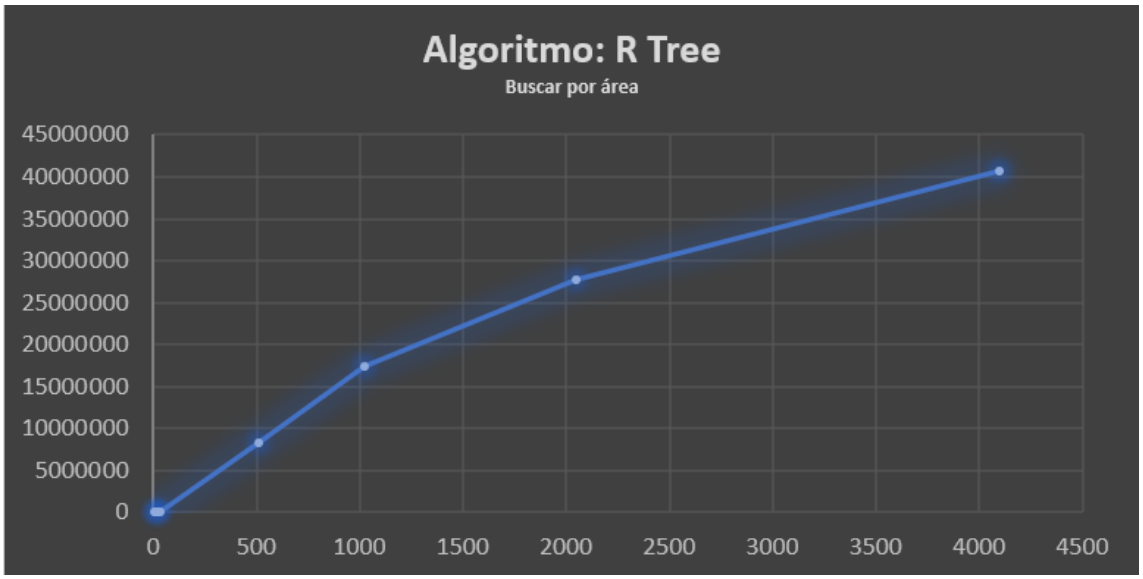
	XXS	XS	S	PERSONALIZAD(	PERSONALIZAD(	PERSONALIZAD(	PERSONALIZAD(
TAMAÑO	8	16	32	512	1024	2048	4096
PRUEBA1	62200	74100	102500	3741100	6966700	21377700	46808400
PRUEBA2	71200	80400	81400	5767700	7067700	21325700	47917500
PRUEBA3	72100	82400	101300	3157300	6532500	22509000	48367800
MEDIA	68500	78966,6667	95066,6667	4222033,333	6855633,333	21737466,67	47697900



El coste de eliminar en un RTree es O(n). Esto lo podemos ver basándonos en la trayectoria que sigue el gráfico, la cual es lineal y augmenta proporcionalmente al tamaño de los ficheros, a más bardizas más tiempo. Resaltar que este coste es el mismo que el de eliminar en el AVLTree, ya que en ambos casos se ha de recorrer todo el árbol para encontrar y eliminar el nodo. En el caso de los AVLTree porque no se elimina por peso (si fuese por peso seria O(log(n))) si no por ID, y por tanto si o sí se ha de buscar en todo el AVLTree, de la misma forma que el RTree.

2.3.4.3. Resultados de buscar por área

	XXS	XS	S	PERSONALIZAD(	PERSONALIZAD(	PERSONALIZAD(	PERSONALIZAD(
TAMAÑO	8	16	32	512	1024	2048	4096
PRUEBA1	30300	34900	103500	8173300	10812500	27364600	44364200
PRUEBA2	30600	36900	70200	8291200	10490300	29117800	36355800
PRUEBA3	30300	34100	75600	8367100	10107000	26711400	41518300
MEDIA	30400	35300	83100	8277200	10469933,33	27731266,67	40746100



Al observar la gráfica, podríamos inicialmente tener dudas sobre si el coste es O(n) o O(log(n)). Sin embargo, al examinarla más detenidamente, se puede apreciar que la tendencia de la gráfica sugiere claramente un coste de O(log(n)).

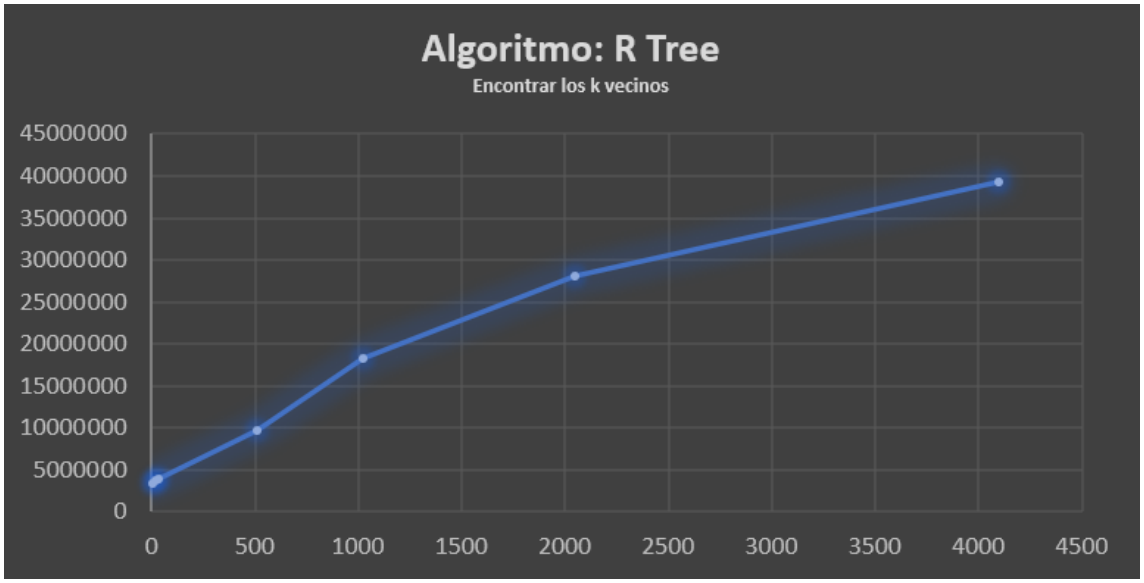
La razón por la que esperamos un coste de O(log(n)) en este caso es debido a la naturaleza de la estructura de datos que estamos utilizando: un R-tree. Los R-trees son árboles de búsqueda balanceados, lo que significa que se mantienen de manera que la profundidad de todas sus hojas sea aproximadamente la misma. Cuando realizamos una búsqueda en un R-tree, no necesitamos buscar en todos los nodos del árbol. Esto se debe al hecho de que la búsqueda puede "descender" por el árbol, pudiendo seguir solo las ramas que corresponden a nodos cuyos rectángulos delimitadores mínimos (MBR) se superponen con el área de búsqueda. Esto significa que la búsqueda puede ignorar una gran cantidad de nodos que no son relevantes para la consulta, lo que resulta en un coste de búsqueda de O(log(n)) en el caso promedio.

En contraste, un coste de O(n) se representaría como una línea recta, como se observó en el caso anterior de la eliminación de la bardiza.

Por lo tanto, al analizar el resultado, podemos confirmar que es consistente con nuestras expectativas teóricas. La tendencia observada en la gráfica confirma que nuestro algoritmo está funcionando de acuerdo con esta complejidad teórica esperada.

2.3.4.4. Resultados de encontrar los k vecinos

	XXS	XS	S	PERSONALIZADO	PERSONALIZADO	PERSONALIZADO	PERSONALIZADO
TAMAÑO	8	16	32	512	1024	2048	4096
PRUEBA1	3229200	3867500	3818200	10603300	22694900	28064600	39930700
PRUEBA2	3533600	3620900	4076300	7840300	18603800	29117800	47388200
PRUEBA3	3418600	3672600	3724600	10622200	13687600	27138200	42339300
MEDIA	3393800	3720333,33	3873033,33	9688600	18328766,67	28106866,67	43219400



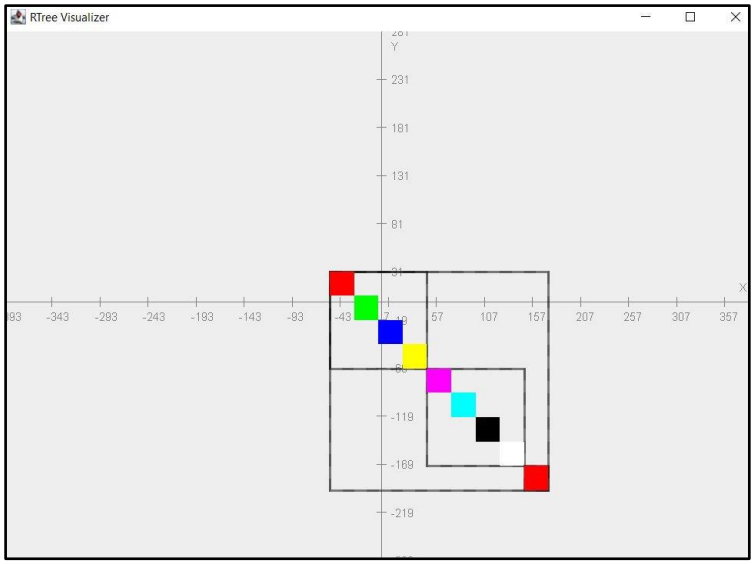
De la misma forma que la función anterior de búsqueda por área, podemos observar un coste O(log(n)). Reiteramos que esta gráfica podría causar cierta confusión con un coste O(n), pero si nos fijamos detenidamente y comparamos la tendencia de una gráfica con coste O(n) como la del apartado de eliminar Bardiza (2.3.4.2), vemos que en efecto es logarítmica. Por tanto, podemos afirmar que la implementación es correcta ya que concuerda con los costes teóricos esperados. Buscar los k vecinos al fin y al cabo no deja de ser una búsqueda en el R-tree, y como tal, se puede beneficiar de las propiedades de esta estructura de datos. Al igual que con la búsqueda por área, la búsqueda de los k vecinos más cercanos puede ignorar una gran cantidad de nodos que no son relevantes para la consulta, lo que resulta en un coste logarítmico en el caso promedio. Por lo tanto, la tendencia observada en la gráfica confirma la eficiencia esperada de nuestro algoritmo para esta operación.

2.3.5. Método de pruebas utilizado

Para garantizar la correcta implementación y funcionamiento de nuestro RTree, empleamos tres métodos de prueba detallados a continuación:

- **Herramienta de depuración de IntelliJ IDEA**: Esta herramienta nos permitió inspeccionar el estado interno del programa en tiempo real, incluyendo el valor de las variables y la estructura del árbol en cualquier punto durante la ejecución. Además, la capacidad de rastrear la ejecución paso a paso nos proporcionó una visión detallada de cómo se desarrollaban las operaciones y cómo interactuaban las diferentes partes del código. Esta información fue invaluable para identificar y corregir errores en la implementación.
- **Visualización en consola de rectángulos y bardizas**: Implementamos una representación textual de la estructura del árbol, mostrando todos los rectángulos y sus respectivas bardizas asociadas. Esta representación nos permitió verificar rápidamente la correcta asignación de los nodos y la integridad de la estructura del árbol. Además, nos proporcionó una forma sencilla de rastrear las operaciones de inserción y búsqueda, facilitando la identificación de cualquier comportamiento inesperado.
- **Visualización gráfica con Swing**: Intentamos representar gráficamente el RTree utilizando la biblioteca Swing de Java. Sin embargo, debido a la proximidad de los puntos en los conjuntos de datos, encontramos que los nodos se superponían en la

visualización. Para superar este obstáculo, introdujimos manualmente bardizas con distancias mayores entre ellas. Aunque esta no es una representación exacta de los datos reales de los datasets, nos permitió confirmar que la estructura como tal almacenaba y mostraba correctamente las bardizas. Esta adaptación, aunque no ideal, nos proporcionó una confirmación visual adicional de la correcta implementación de nuestro RTree. Añadir que finalmente mostramos los datasets por consola, debido a que se nos superponían todos los puntos y no conseguimos solucionarlo.



2.3.6. Problemas observados

Como mencionábamos en el apartado anterior, a la hora de visualizar el árbol para verificar su correcto funcionamiento, tuvimos serios problemas con Swing. Ya que como las bardizas de los datasets están extremadamente juntas entre ellas, a razón del noveno o décimo decimal, nos proporcionó serios problemas que no conseguimos solucionar. Es por ello, que finalmente nos decantamos por mostrarlo por consola con todas sus respectivas conexiones Rectángulo-Nodos, ya que era más simple. Aun así, planteamos como solución, reducir todas las coordenadas de las bardizas mediante una formula, para normalizarlas y poderlas dibujar más separadas. Esta idea la intentamos poner en marcha, sin logro alguno, pero creemos que con más tiempo hubiésemos podido lograrlo.

2.4. Tablas de Hash

2.4.1. Introducción

Las tablas de hash son estructuras de datos que permiten almacenar y recuperar información utilizando una función de hash para generar una dirección de memoria única a partir de una clave. Gracias a esta función, las tablas de hash ofrecen tiempos de búsqueda, inserción y eliminación casi constantes en promedio. Estas estructuras son ampliamente utilizadas en aplicaciones donde se requiere un acceso rápido a la información, como en la implementación de diccionarios, cachés y sistemas de detección de duplicados.

2.4.2. Diseño

Para el diseño de las tablas de hash, hay numerosos desafíos asociados con su implementación, particularmente en relación con la resolución de colisiones, el manejo del factor de carga y la

distribución uniforme de las claves. Nuestro diseño de HashTable aborda estos desafíos de manera efectiva mediante el uso de diversas técnicas y enfoques optimizados:

- **Uso de dos funciones hash:** El "Double Hashing" es una técnica de resolución de colisiones que utiliza dos funciones hash en lugar de una. La primera función hash se usa para obtener la ubicación inicial en la tabla hash, y si hay una colisión, se usa la segunda función hash para determinar un salto para la siguiente ubicación disponible. Esta técnica es efectiva para manejar las colisiones, ya que reduce la probabilidad de agrupación secundaria, una situación en la que las colisiones tienden a agruparse en ciertas áreas de la tabla hash, lo que aumenta el tiempo de búsqueda promedio.
- **Resizing dinámico:** En una tabla hash, el factor de carga (la cantidad de entradas ocupadas en la tabla dividida por el tamaño total de la tabla) juega un papel crucial en su rendimiento. A medida que el factor de carga se acerca a 1, el número de colisiones aumenta, lo que lleva a un mayor tiempo de búsqueda. Para evitar este problema, se implementa una operación de resizing, que duplica el tamaño de la tabla en el doble y que se activa cuando la tabla hash alcanza un cierto factor de carga (en este caso, el 80%). Al duplicar el tamaño de la tabla hash y redistribuir las entradas existentes, se mantiene el factor de carga bajo control, asegurando que la tabla hash mantenga su rendimiento.
- **Uso de listas enlazadas para manejar colisiones:** Esta técnica se llama direccionamiento abierto, y es una técnica en la que se manejan las colisiones permitiendo que más de un elemento ocupe una ubicación en la HashTable. En este caso, cada ubicación en la HashTable es una lista enlazada de elementos "LinkedList<Accussed>", lo que permite almacenar varios elementos con la misma clave hash en la misma ubicacióN. Esta es una forma muy eficiente de manejar las colisiones, ya que permite un número teóricamente ilimitado de colisiones por ubicación de hash, limitado solo por la cantidad de memoria disponible.
- **Selección del tamaño inicial de la tabla como número primo:** Los números primos son útiles en el hash porque ayudan a evitar patrones que pueden surgir con ciertos conjuntos de claves, y especialmente en las operaciones de modulación, tienden a proporcionar una distribución más uniforme de las claves. En este caso, el tamaño de la tabla hash se elige un número primo para minimizar las colisiones y garantizar una distribución uniforme de las claves.
- **Estructura de datos genérica:** Nuestra HashTable se implementa de una manera que puede manejar cualquier tipo de datos, no solo números o cadenas. En este caso, se está utilizando para almacenar objetos personalizados de tipo "Accussed", lo que demuestra su versatilidad. Esto también significa que se puede reutilizar esta HashTable en diferentes contextos simplemente cambiando el tipo de datos que se están almacenando.

2.4.3. Algoritmos implementados

2.4.3.1. Insertar acusados en la tabla

Una vez recogida toda la información, la función crea un nuevo objeto Accused con el nombre, el número de conejos y la profesión proporcionados, la función añade el objeto a la tabla hash utilizando la función addAccused() de la clase HasgTable. Esta función utiliza un método conocido como doble dispersión (double hashing) para tratar las colisiones que se producen cuando dos objetos tienen el mismo valor de hash. A continuación, se describe el proceso en detalle:

- **Cálculo de las funciones de hash:** Para cada objeto `Accused` se calculan dos valores de hash utilizando dos funciones de hash distintas: `firstHashFunction` y `secondaryHashFunction`. Estas funciones toman como entrada el nombre del acusado y devuelven un valor de hash.
- **Obtención del índice:** El índice en la tabla hash donde se desea colocar el acusado inicialmente se obtiene utilizando el valor del primer hash (`primaryHash`).En caso de colisiones, se ha creado un bucle `for` que verifica si la posición de la tabla hash indicada por `newIndex` está vacía. Si no lo está (es decir, hay una colisión), calcula un nuevo índice utilizando la función `doubleHashing`. En este caso la función teórica es la siguiente, h (k, i) = (h1(k) + h2(k) * i) % R que en el código tomamos como parámetros el valor devuelto por `primaryHash`, h1(k), `secondaryHash` h2(k) y `i` (el número de intento). Este proceso se repite hasta que se encuentra una posición vacía.
- **Inserción del acusado:** Una vez encontrada una posición vacía en la tabla hash, se añade el acusado a esa posición utilizando el método `add` de la lista en esa posición.
- **Aumento del conteo de elementos:** Se incrementa el contador de elementos en la tabla hash (`elements`) y se verifica si el número de elementos en la tabla hash ha alcanzado o superado el 80% de su tamaño. Si es así, se redimensiona la tabla hash para evitar un rendimiento deficiente y altas tasas de colisiones. El método `resizeHashTable(true)` se encarga de este redimensionamiento, creando una tabla más grande y redistribuyendo los elementos en la nueva tabla.

2.4.3.2. Eliminar acusados en la tabla

La función para eliminar acusados es una función miembro de la clase HashTable, cuya finalidad es eliminar a una persona acusada de la tabla hash por su nombre. La función primero calcula el código hash del nombre utilizando los métodos firstHashFunction() y secondHashFunction(). Luego, el código hash previamente calculado se usa para encontrar el índice de la persona acusada en la tabla hash. Si se encuentra a la persona acusada, la función la elimina de la tabla hash y devuelve true como parámetro de salida, para de esta manera hacer saber al Manager que debe mostrar el mensaje de eliminación exitosa. De lo contrario, la función devuelve falso y muestra un error por consola.

La función de eliminación sigue los siguientes pasos:

- La función se declara y toma una cadena como entrada.
- El código hash del nombre se calcula utilizando los métodos firstHashFunction() y secondHashFunction().
- El índice de la persona acusada en la tabla hash se encuentra usando el código hash.
- Si se encuentra a la persona acusada, se elimina de la tabla hash.
- La función devuelve verdadero si la persona acusada se elimina de la tabla hash, o falso en caso contrario.

2.4.3.3. Marcar como herejía

La función “markAsHeretic()” es responsable de buscar un objeto de tipo “Accused” por su nombre y, si se encuentra, actualizar su estado de herejía a “true” o “false”, dependiendo del valor del parámetro “isHeretic”.

- **Cálculo de las funciones de hash:** Al igual que con la función “addAccused()” y “deleteAccussed()”, aquí también calculamos dos valores de hash utilizando las mismas funciones de hash: “firstHashFunction()” y “secondaryHashFunction()”.
- **Obtención del índice:** El índice en la tabla hash donde se desea buscar el acusado inicialmente se obtiene utilizando el valor del primer hash (“primaryHash”).
- **Manejo de colisiones y búsqueda:** Si la lista en la posición “newIndex” no está vacía, buscamos al acusado con el nombre dado en esa posición. Si el primer acusado en esa lista tiene el nombre que buscamos, hemos encontrado al acusado y podemos proceder al siguiente paso. Si no, calculamos un nuevo índice usando la función “doubleHashing()” y repetimos el proceso hasta que encontremos al acusado o recorramos todas las posiciones de la tabla.
- **Marcado como hereje:** Una vez que encontramos al acusado, verificamos su profesión. Si no es un REY, una REINA o un CLÉRIGO, entonces podemos cambiar su estado de hereje al valor de “isHeretic”. Si es alguna de estas tres profesiones, mostramos un mensaje de error y no cambiamos su estado de hereje. Y en el supuesto caso de que remarquemos un Hereje que ya era Hereje, simplemente lo sobreescribimos y ya está, mostrando un el mensaje correspondiente por consola.

2.4.3.4. Juicio final (usuario en específico)

La función “finalJudgementOneAccused()” es responsable de buscar un objeto de tipo “Accused” por su nombre y, si se encuentra, imprimir su información relevante.

Al igual que en las funciones previamente explicadas, aquí también se calculan dos valores de hash utilizando las mismas funciones de hash: “firstHashFunction()” y “secondaryHashFunction()”. En caso de colisión, calculamos un nuevo índice utilizando la función “doubleHashing()” y repetimos el proceso hasta que encontremos al acusado o recorramos todas las posiciones de la tabla.

Una vez que encontramos al acusado, imprimimos su nombre, la cantidad de conejos que ha visto, su profesión y si es o no un hereje. Si no encontramos al acusado después de haber comprobado todas las posiciones posibles en la tabla hash, mostramos un mensaje de error indicando que no se ha encontrado ningún acusado con ese nombre.

2.4.3.5. Resultado del juicio final (rango de valores)

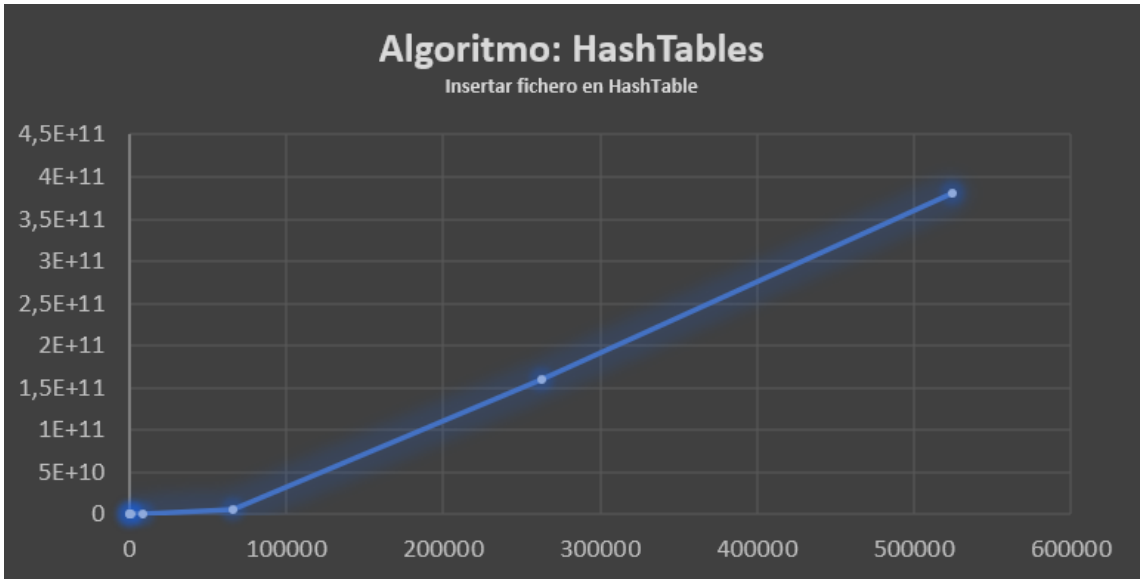
La función “finalJudgementAllAccused()” en la clase “HashTable” se encarga de buscar a todos los acusados cuyo número de conejos vistos cae dentro del rango especificado por “minRabbits” y “maxRabbits”, e imprimir su información relevante.

Se recorre toda la tabla hash. Para cada lista vinculada en la tabla, se revisan todos los acusados. Si un acusado ha visto un número de conejos que se encuentra entre “minRabbits” y “maxRabbits” (ambos inclusive), se agrega a la “accusedList”. Después de haber revisado toda la tabla hash, se verifica si “accusedList” está vacía. Si lo está, significa que no se encontró ningún acusado que haya visto un número de conejos entre “minRabbits” y “maxRabbits”, y se imprime un mensaje de error.

2.4.4. Análisis de rendimiento y resultados

2.4.4.1. Resultados de insertar todos los acusados de los ficheros

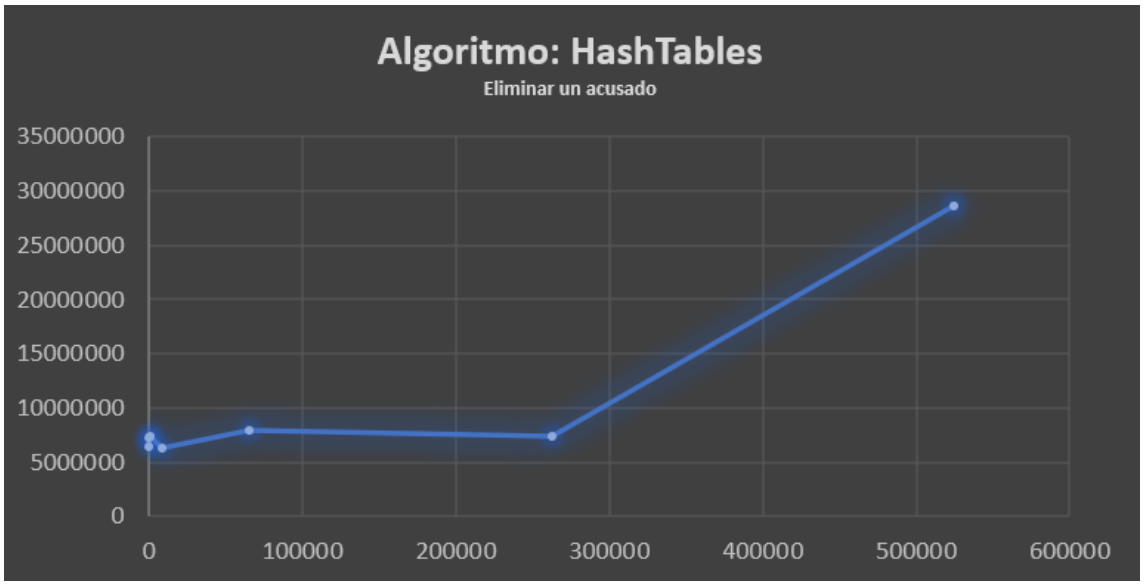
	XXS	XS	S	M	L	XL	XXL
TAMAÑO	8	16	512	8192	65536	262144	524288
PRUEBA1	1044500	1639899	4586900	25776400	6192457700	1,35976E+11	3,90571E+11
PRUEBA2	1143600	1114000	4742200	26409999	5957120500	1,67272E+11	4,12272E+11
PRUEBA3	1391600	1240700	4573500	25805599	6027617599	1,78841E+11	3,41442E+11
MEDIA	1193233,33	1331533	4634200	25997332,67	6059065266	1,60696E+11	3,81428E+11



Bien implementado, el coste teórico debería ser O(1), sin embargo, el practico puede llegar a incrementar hasta O(N) donde N sería el número de elementos de la tabla, este coste también sería el peor caso ya que habría que recorrer todos los elementos para poder insertar el fichero, y en el mejor caso sería coste O(1) ya que la primera interacción sería la correcta.

2.4.4.2. Resultados de eliminar un acusado

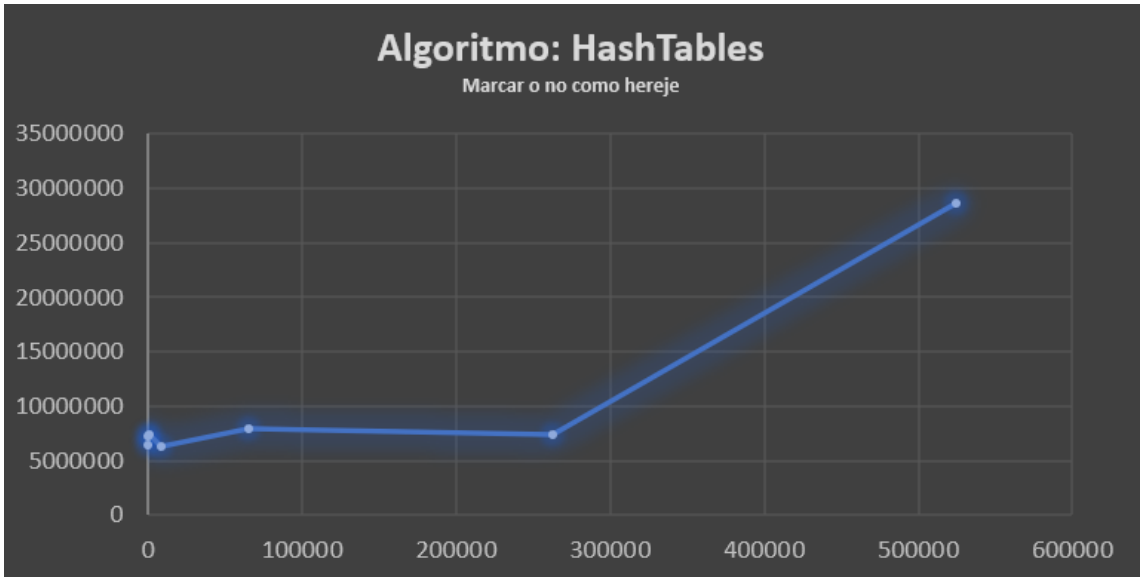
	XXS	XS	S	M	L	XL	XXL
TAMAÑO	8	16	512	8192	65536	262144	524288
PRUEBA1	6977800	6680600	7241199	6152301	8281200	5851100	51938000
PRUEBA2	6455000	5733600	6098100	6416201	6890200	8445601	23844500
PRUEBA3	8528901	6822300	8935200	6303101	8640600	7782291	10200800
MEDIA	7320567	6412166,67	7424833	6290534,333	7937333,333	7359664	28661100



En cuanto a la eliminación, el coste teórico y práctico son los mismos que en el apartado anterior ya que eliminar un acusado accediendo directamente y sin generar colisiones sería un coste O(1) y el peor caso donde todo colisionara sería coste O(n) siendo N el tamaño de elementos de la tabla.

2.4.4.3. Resultados de marcar como hereje

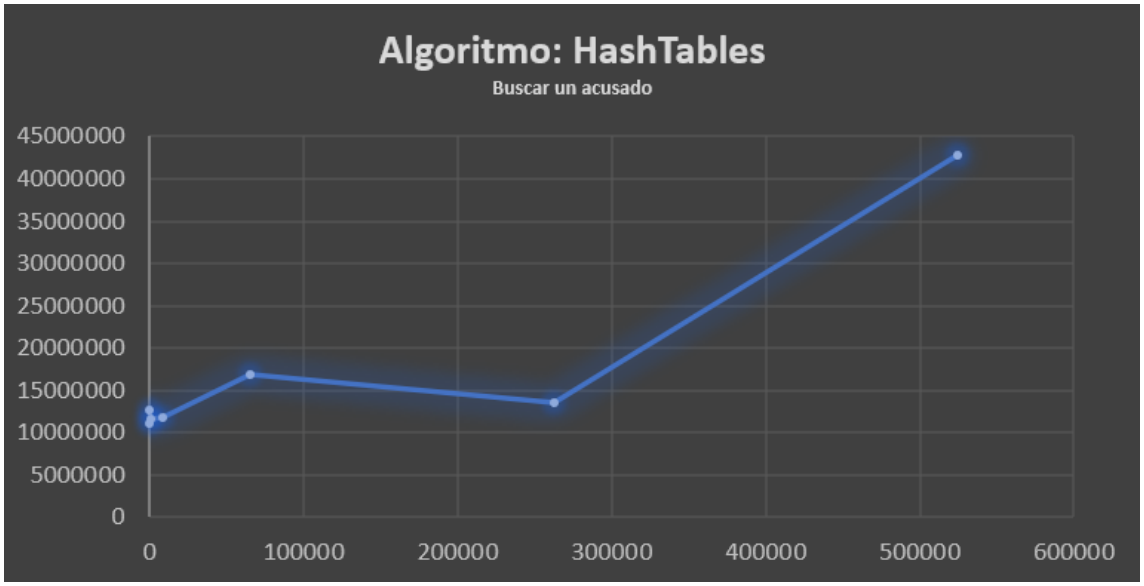
	XXS	XS	S	M	L	XL	XXL
TAMAÑO	8	16	512	8192	65536	262144	524288
PRUEBA1	4827300	4673700	4190000	3806700	7346200	3737200	18157500
PRUEBA2	4298300	3946400	4758000	3840600	7357900	4518700	10564700
PRUEBA3	3957500	4689300	3753300	5115800	7528100	4507500	10216100
MEDIA	4361033,33	4436466,67	4233766,67	4254366,667	7410733,333	4254466,667	12979433,33



El coste podemos observar que al inicio es constante, ya que la cantidad de acusados en el fichero es muy pequeña, pero a medida que se incrementa significativamente la cantidad de Acusados es más difícil que el acusado buscado para marcar como hereje esté al inicio de la tabla siendo coste O(1), es por eso que al buscar entre N acusados del fichero, el coste tiende a ser coste O(N). Aún así el coste es correcto y por tanto la HashTable podemos afirmar que está bien implementada, ya que la teoría dada en clase afirmaba estos costes: O(1) en el mejor caso y O(N) en el peor de los casos.

2.4.4.4. Resultados de buscar un acusado

	XXS	XS	S	M	L	XL	XXL
TAMAÑO	8	16	512	8192	65536	262144	524288
PRUEBA1	12734500	11081900	11667300	11707800	16807400	13522800	42826800

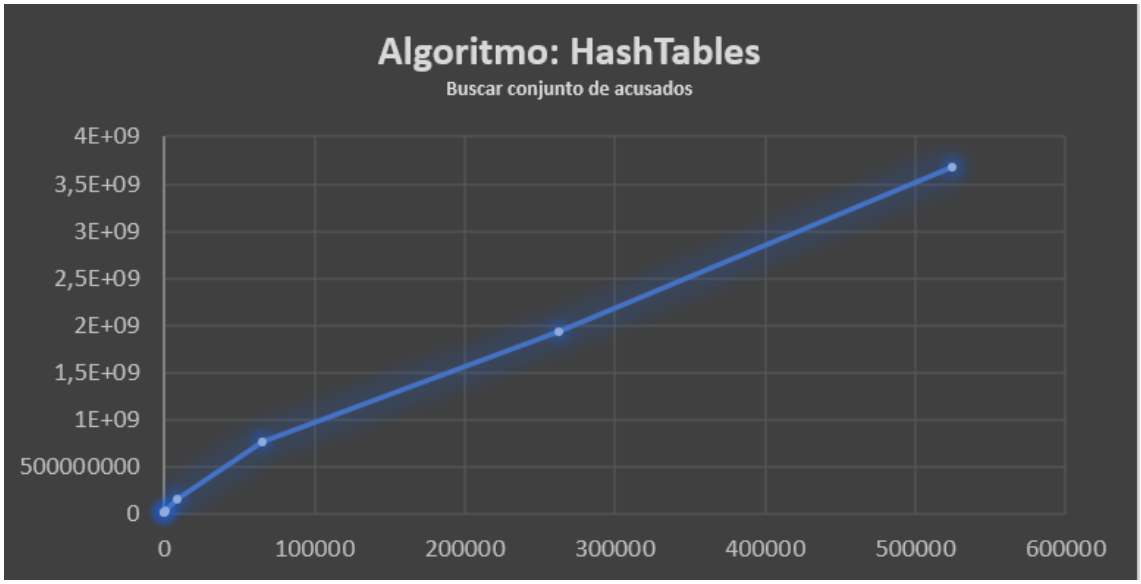


Como podemos observar, al buscar un acusado de forma aleatoria, no podemos determinar el coste exacto, es por eso, que la gráfica hacia una forma al principio constante y al final con coste(N). Esto se debe a que buscamos un acusado aleatorio que puede estar tanto en la primera, como en el medio o al final de la tabla, siendo en el mejor de los casos coste constante de O(1) y en el peor de los casos coste O(N), ya que en el peor de los casos lo ha de buscar entre N acusados que tiene el fichero.

2.4.4.5. Resultados de buscar un conjunto de acusados

NOTA: Para calcular estos resultados, hemos empleado un número de conejos entre 1 y 10.000.000 para mostrar de manera intencionada todos los acusados de la HashTable, y de esta manera ver de manera más clara la tendencia de la gráfica.

	XXS	XS	S	M	L	XL	XXL
TAMAÑO	8	16	512	8192	65536	262144	524288
PRUEBA1	16046000	13935300	33558800	160957900	763990800	1942334300	3673725900

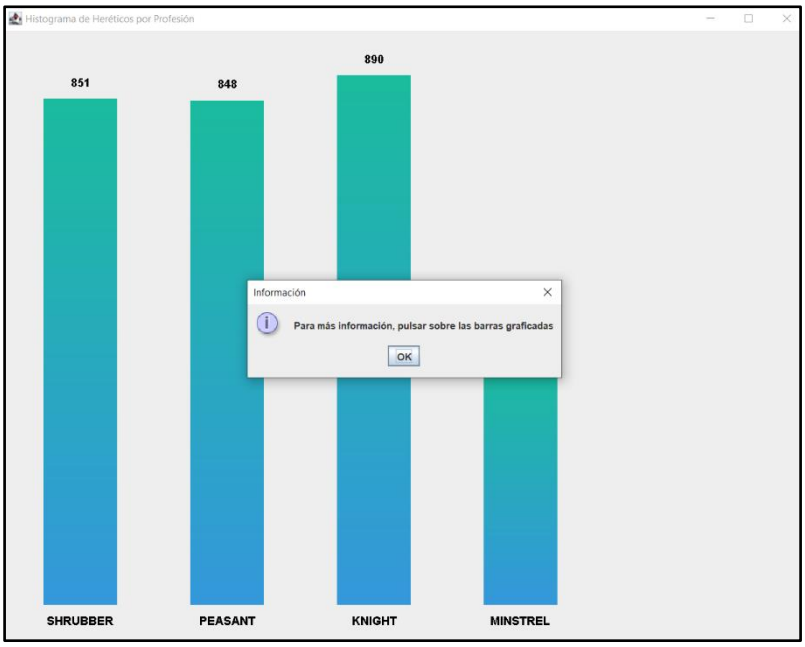


Podemos observar que el coste en el peor de los casos de todos los ficheros es coste O(N). Esto se debe a que de manera intencionada hemos buscado todos los acusados del fichero, y como en cada fichero son N acusados, esto hace que el coste sea O(N). Sabiendo que este es el peor

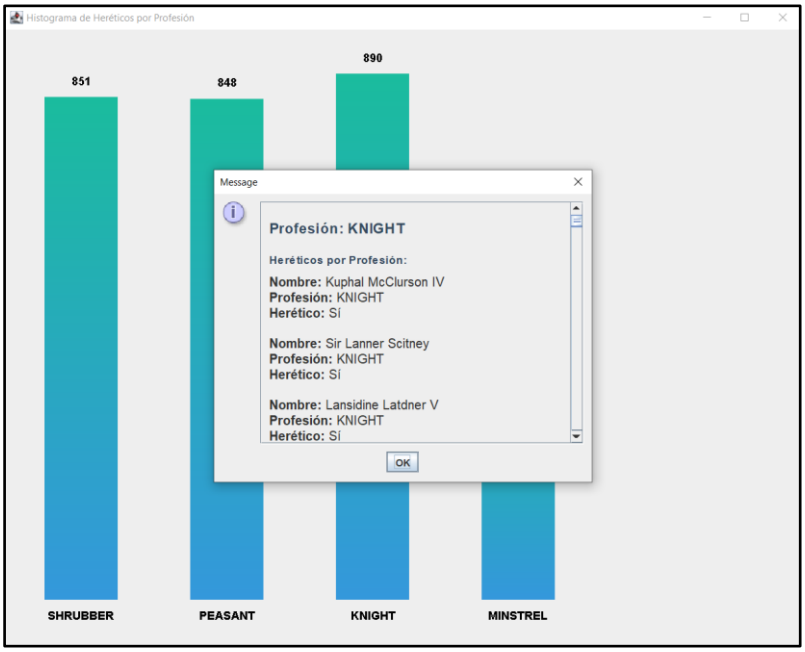
caso, solo nos queda resaltar que en el supuesto caso de buscar un rango de conejos que solo el primer acusado del fichero tiene, el coste seria O(1) y sería el mejor caso tanto teórico como práctico.

2.4.5. Método de pruebas utilizado

En cuanto al método de pruebas hemos usado la herramienta debugger de IntelliJ IDEA, con ella hemos podido encontrar errores en el proceso de creación de las tablas tanto en el apartado de inserción como los demás. También hemos podido comprobar los resultados mostrándolos con Java/Swing, lo que ha permitido poder ver errores de manera muy visual, pudiendo así corregir ciertos fallos en cuanto a la ejecución de código. Adjuntamos imagen:



Y al pulsarlo podemos corroborar si los ficheros se leen bien, sobretodo lo comprobamos con los ficheros XXS y S ya que eran los más pequeños y fáciles de comprobar:



De esta manera ha sido muy fácil comprobar y debuggar el correcto funcionamiento de nuestra HashTable. Sabemos que la visualización se podría mejorar y hacer más afable para el usuario, pero como cumple los requisitos mínimos y ya nos era más que suficiente para debuggar, decidimos dejarla como está.

2.4.6. Problemas observados

Uno de los principales problemas que hemos tenido han sido las colisiones, tanto incorporando elementos a la tabla como eliminándolos. También elegimos un numero primo como tamaño de la tabla, pero esto nos dio más problemas que ayudarnos, para empezar, desperdiciábamos espacio de la tabla, lo cual afectó levemente en el rendimiento en la inserción y eliminación. Otro problema que tuvimos fue el redimensionamiento, cuando intentábamos ajustar el factor de carga, teníamos problemas para encontrar el siguiente número primo y para ello necesitamos cálculos adicionales. El ultimo inconveniente que se nos presento fue al momento de multiplicar por 2 la capacidad de la tabla cuando detectábamos que llegaba a un 80% de la capacidad, en el momento de recalcular todo, se nos generaban colisiones ya que no habíamos implementado del todo bien el proceso de rehashing.

3. Análisis estructuras de datos auxiliares

3.1. Arraylist

ArrayList es una estructura ya implementada en Java que implementa la interfaz “List” que permite almacenar objetos de forma dinámica.

Su capacidad se ajusta automática e internamente cuando superamos su tamaño inicial. Entre sus operaciones fundamentales se encuentran 'add', 'get' y 'remove'. 'Add' y 'get' normalmente presentan un costo computacional de O(1), ya que añaden o recuperan elementos al final de la lista.

Sin embargo, si se inserta o elimina un elemento en una posición específica o si se requiere duplicar toda la información, la complejidad se eleva a O(n). Este tipo de estructura nos proporcionan una gran flexibilidad además de ser muy utilizadas en aplicaciones que requieren manipulación frecuente de datos.

3.2. Hashmap

HashMap es otra estructura que ya viene implementada en Java e implementa la interfaz Map. Permite el almacenamiento de datos en pares clave-valor, donde cada clave es única.

Su diseño optimiza la recuperación de datos, ya que la complejidad computacional para las operaciones básicas como agregar, obtener o eliminar un elemento es normalmente O(1).

Sin embargo, en escenarios peores, como cuando se producen colisiones de claves, la complejidad puede llegar a ser O(n). Los HashMaps son fundamentales en aplicaciones que requieren búsquedas rápidas y eficientes, como las bases de datos y los motores de búsqueda

3.3. Queue

Queue es la tercera y última interfaz que hemos utilizado, y de nuevo viene implementada en Java. Esta representa una estructura de datos de tipo FIFO (First In, First Out). Esto significa que los elementos se eliminan en el mismo orden en que se agregan.

Las operaciones fundamentales 'add', 'remove' y 'getFirst' tienen un costo de O(1), lo que las hace particularmente eficientes para aplicaciones como el manejo de solicitudes en tiempo real, la planificación de tareas o incluso algoritmos de búsqueda y recorrido.

En muchos casos, se emula el comportamiento de Queue utilizando otras clases como ArrayList, lo que puede incrementar el costo computacional.

4. Conclusiones

A lo largo de este curso, hemos explorado profundamente el lenguaje de programación Java y varias de sus aplicaciones, incluyendo el desarrollo e implementación de diversas estructuras de datos no lineales como grafos, árboles AVL, R-trees y tablas de hash.

El lenguaje de programación Java fue escogido por sus ventajas, tales como su robustez, seguridad, portabilidad y su amplia aceptación en la comunidad de desarrolladores. Sin embargo, como todo lenguaje, también tiene sus desventajas, como su alto consumo de memoria y el tiempo de ejecución, que en algunos casos puede ser mayor que en otros lenguajes. A pesar de esto, Java resultó ser una herramienta poderosa y versátil para nuestro propósito.

La utilización de las estructuras de datos no lineales nos permitió entender la importancia de estas en diferentes ámbitos de la informática. Los grafos con matrices de adyacencia, por ejemplo, demostraron su utilidad en el modelado y resolución de problemas complejos en áreas como la teoría de redes y la optimización. A pesar de ciertos desafíos encontrados, logramos implementar algoritmos y realizar análisis de rendimiento que evidenciaron su eficacia.

Los árboles AVL y R-trees, por su parte, fueron de gran importancia para entender cómo la autogestión del balanceo puede optimizar las operaciones de búsqueda, inserción y eliminación. En este sentido, se comprobó que la eficiencia de estas estructuras es de vital importancia en aplicaciones de bases de datos y sistemas de archivos.

Las tablas de hash nos permitieron apreciar cómo una buena función de hash puede facilitar la búsqueda y almacenamiento de elementos de manera eficiente. Pudimos experimentar cómo el manejo apropiado de colisiones y el redimensionamiento pueden impactar en el rendimiento de estas estructuras.

Cada estructura de datos presentó sus propios desafíos y problemas, pero la solución de estos nos permitió aprender y crecer como programadores. La implementación de pruebas fue un aspecto crucial en este proceso, nos ayudó a identificar errores, validar soluciones y comprender mejor el comportamiento de nuestras estructuras de datos.

A través de este curso, no solo hemos aprendido a programar en Java, sino también hemos adquirido una comprensión más profunda y apreciación por la complejidad y belleza de las estructuras de datos no lineales. Estas lecciones y experiencias nos equipan con las habilidades y el conocimiento necesarios para enfrentar problemas más desafiantes en el futuro, y nos motivan a seguir explorando y aprendiendo en el amplio campo de la informática.

5. Bibliografía

GeeksforGeeks. (n.d.). "Data Structures". Recuperado de: https://www.geeksforgeeks.org/data-structures/

Tutorialspoint. (n.d.). "Data Structures and Algorithms". Recuperado de: https://www.tutorialspoint.com/data_structures_algorithms/index.htm

Programiz. (n.d.). "Binary Tree". Recuperado de:

https://www.programiz.com/dsa/binary-tree

Wikipedia. (n.d.). "Binary Tree". Recuperado de: https://en.wikipedia.org/wiki/Binary_tree

Computer Science Unleashed. (n.d.). "Data Structures: Binary Trees". Recuperado de: https://www.csunplugged.org/en/topics/data-structures/

Edureka. (n.d.). "Binary Trees in Java: Everything You Need to Know". Recuperado de: https://www.edureka.co/blog/binary-tree-in-java/

LeetCode. (n.d.). "Explore - Binary Tree". Recuperado de: https://leetcode.com/explore/learn/card/data-structure-tree/