Práctica Capítulo 1.- Análisis de la eficiencia de tres algoritmos para encontrar la moda de un vector de enteros positivos (2023)

Juan Francisco Hernández Fernández
Marco Martínez González
Pere Marc Monserrat Calbo
Jordi Sevilla Marí

[1]

***Resumen*—En este documento se explica nuestra implementación de la práctica "Análisis de la eficiencia de tres algoritmos para encontrar la moda de un vector de enteros positivos". En este, explicamos como hemos implementado el patrón MVC, el funcionamiento de la interfaz gráfica de nuestro programa, el funcionamiento de nuestros algoritmos de ordenación, la estructura de los datos utilizada para solucionar el problema e información sobre las particularidades del funcionamiento y una guía de uso.**

***Términos del Índice*—** Coste asintótico, Instrucción crítica , Moda, Patrón MVC, Patrón por eventos.

I. INTRODUCCIÓN

En este primer taller se desarrolla con patrón MVC una aplicación que presenta una gráfica de los costes computacionales asintóticos de tres algoritmos. Los algoritmos buscan la moda de un vector numérico que contiene valores enteros positivos aleatorios. Se hacen diferentes pruebas variando la N del vector (N=Tamaño de éste). Los dos primeros encuentran la moda con un coste asintótico entre O(n) y O(n·log n). El tercero implementa el producto vectorial del mismo vector por sí mismo, representando así un coste $O(n^2)$.
La presente memoria se acompaña del proyecto del programa Java. Además de un vídeo de unos 10 minutos, donde se muestra cómo es la distribución del código, cómo se compila el proyecto y se ejecuta y se explican los trabajos realizados.

II. ENTORNO DE PROGRAMACIÓN EMPLEADO

Para la implementación de la práctica se ha utilizado Apache NetBeans IDE 12.5.

III. CONCEPTOS DEL PATRÓN MODELO VISTA CONTROLADOR

El patrón de arquitectura de software Modelo-Vista-Controlador (MVC) se utiliza comúnmente en el desarrollo de aplicaciones de software. El modelo MVC divide una aplicación en tres componentes principales: el modelo, la vista y el controlador. Cada uno de estos componentes tiene una función específica y juntos, proporcionan una forma eficiente y escalable para desarrollar aplicaciones de software. A continuación, se presentan algunas ventajas del uso del modelo MVC en la programación informática:

1. Separación de responsabilidades: El modelo MVC divide las tareas de una aplicación en tres componentes distintos, cada uno de los cuales tiene una responsabilidad específica. El modelo maneja los datos y la lógica de negocios, la vista es responsable de la presentación de los datos y la interfaz de usuario, y el controlador maneja la entrada del usuario y coordina las acciones entre el modelo y la vista. Esta separación de responsabilidades hace que sea más fácil para los desarrolladores trabajar en cada componente de manera independiente, lo que puede mejorar la calidad del código y reducir los errores.
2. Facilidad de mantenimiento: La separación de responsabilidades también hace que sea más fácil mantener una aplicación MVC. Si necesita cambiar algo en la lógica de negocios, por ejemplo, solo tendrá que hacer cambios en el modelo, sin tener que preocuparse por el efecto en la vista o el controlador. Esto facilita el mantenimiento y la evolución de la aplicación a medida que cambian los requisitos.
3. Reutilización de código: Al separar la lógica de negocios y la presentación de la interfaz de usuario, el patrón MVC permite reutilizar el código en diferentes partes de la aplicación o incluso en diferentes aplicaciones. Por ejemplo, si tiene una aplicación que utiliza el mismo modelo de datos en varias vistas, puede reutilizar el modelo en cada vista en lugar de tener que crear uno nuevo para cada una. Esto reduce la duplicación de código y hace que el desarrollo sea más eficiente.
4. Pruebas unitarias: El patrón MVC facilita las pruebas unitarias. Cada componente de la aplicación se puede probar de forma independiente, lo que facilita la identificación y corrección de errores. Las pruebas de unidad también son más fáciles de automatizar y ejecutar repetidamente, lo que ayuda a garantizar que la aplicación sea estable y confiable.

5. Escalabilidad: El patrón MVC es escalable y permite la adición de nuevas características o módulos sin afectar la funcionalidad existente. El uso de un modelo de datos separado también permite la escalabilidad horizontal, lo que significa que se pueden agregar más servidores o recursos para aumentar la capacidad de la aplicación.

En general, el uso del patrón MVC en la programación informática puede mejorar la calidad del código, facilitar el mantenimiento y la evolución de la aplicación, mejorar la reutilización del código, facilitar las pruebas unitarias y permitir la escalabilidad.

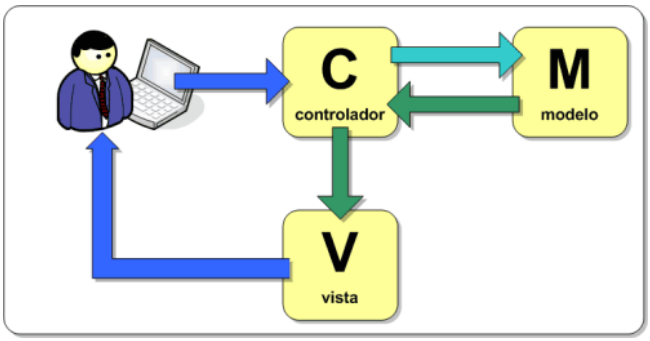

Fig. 1. Patrón MVC

IV. Patrón por eventos

El patrón por eventos es un patrón de diseño de software utilizado en la programación de aplicaciones que se basan en eventos y notificaciones. Este patrón se basa en la idea de que los objetos en un programa pueden enviar y recibir eventos, y los demás objetos pueden responder a estos eventos de manera apropiada.

En términos simples, el patrón por eventos consiste en establecer un sistema de notificación en el que un objeto, conocido como emisor, envía una señal de evento a uno o varios objetos, conocidos como oyentes o suscriptores. Estos objetos oyentes pueden responder a la señal de evento tomando una acción o ejecutando un método determinado.

En nuestro caso, utilizamos el patrón por eventos para nuestra aplicación de escritorio. Cuando un usuario hace clic en el botón “START”, el objeto botón emite un evento de clic. Los objetos suscriptores como el objeto de controlador de eventos están a la espera de este evento y responden a él realizando la búsqueda de la moda en un array de enteros positivos.

El patrón por eventos es un enfoque muy útil para la programación de aplicaciones interactivas y en tiempo real, ya que permite que las aplicaciones respondan de manera rápida y eficiente a los eventos del usuario. También permite una mayor modularidad del código, lo que facilita su mantenimiento y evolución.

V. Cálculo de coste asintótico por la instrucción crítica

El cálculo de coste asintótico por la instrucción crítica es una técnica utilizada en el análisis de algoritmos para determinar el tiempo de ejecución de un algoritmo en el peor de los casos. La instrucción crítica se refiere a la operación dentro de un algoritmo que consume la mayor cantidad de tiempo de ejecución.

En el cálculo de coste asintótico por la instrucción crítica, se determina la cantidad de veces que se ejecuta la instrucción crítica en el peor de los casos en función del tamaño de entrada del problema. Luego, se multiplica la cantidad de ejecuciones por la cantidad de tiempo que tarda en ejecutarse la instrucción crítica. Este valor resultante representa el tiempo de ejecución del algoritmo en el peor de los casos.

Por ejemplo, si tenemos un algoritmo que realiza una búsqueda lineal en una lista de n elementos y la instrucción crítica es la comparación entre dos elementos de la lista, entonces el coste asintótico por la instrucción crítica se puede calcular de la siguiente manera: si el peor de los casos se da cuando el elemento buscado no se encuentra en la lista, entonces la instrucción crítica se ejecutará n veces en el peor de los casos. Si la comparación tarda una cantidad constante de tiempo, digamos c, entonces el tiempo de ejecución del algoritmo en el peor de los casos será de $O(n \cdot C_m)$.

En general, el cálculo de coste asintótico por la instrucción crítica es una técnica útil para evaluar el rendimiento de los algoritmos y comparar diferentes soluciones para un mismo problema en términos de su tiempo de ejecución en el peor de los casos. Sin embargo, es importante tener en cuenta que esta técnica no considera otros factores importantes como el uso de memoria o la complejidad de las operaciones realizadas por el algoritmo.

VI. Algoritmos implementados

La moda es el valor que aparece con mayor frecuencia en un conjunto de datos, y para encontrarla es necesario contar el número de veces que aparece cada valor en el vector. Para ello lo más eficiente computacionalmente es ordenar el vector y luego realizar el conteo. En el caso de usar un hashMap se realiza el conteo a la vez que construimos el mapa.

Los algoritmos que hemos implementado en la práctica para buscar la moda de un vector numérico que contiene valores enteros positivos aleatorios son:

- Hash, con un coste de O(n)
- MergeSort, con un coste de $O(n \cdot \log n)$
- QuickSort, con un coste entre $O(n \cdot \log n)$ y $O(n^2)$
- SelectionSort, con un coste $O(n^2)$
- RadixSort, con un coste O(n)
- El tercero implementa el producto vectorial del mismo vector por sí mismo, representando así un coste $O(n^2)$.

Se hacen diferentes pruebas variando la N del vector (N=Tamaño de éste).

A. *MergeSort*

El coste asintótico de ordenar un array de enteros mediante el algoritmo MergeSort es $O(n \cdot \log n)$, donde "n" es el número

de elementos en el array.

El algoritmo MergeSort es un algoritmo de ordenación que utiliza la técnica de "dividir y conquistar". En resumen, el algoritmo divide repetidamente el array en mitades iguales hasta que cada subarray tiene un solo elemento, y luego fusiona los subarrays ordenados para producir un array ordenado completo. En cada división, el algoritmo tarda O(log n) tiempo y se divide el array en dos partes de igual tamaño. Luego, la fusión de dos subarrays ordenados toma O(n) tiempo. Como el algoritmo divide repetidamente el array en dos mitades iguales hasta llegar a subarrays de tamaño uno, el número total de divisiones es O(log n). Cada división implica la comparación de todos los elementos del array, lo que lleva O(n) tiempo. Entonces, el tiempo total de ejecución es O(n log n).

En resumen, el coste asintótico de MergeSort es de O(n log n), lo que significa que el tiempo de ejecución del algoritmo aumenta proporcionalmente a n y log n en el peor de los casos. Esto hace que MergeSort sea un algoritmo eficiente para ordenar grandes conjuntos de datos.

B. RadixSort

El coste asintótico de ordenar un array de enteros mediante el algoritmo RadixSort es O(d * (n + k)), donde "d" es el número de dígitos en los números, "n" es el número de elementos en el array y "k" es el rango de valores posibles de los elementos.

El algoritmo RadixSort es un algoritmo de ordenación que utiliza la técnica de "ordenación por clave". En lugar de comparar los elementos del array entre sí, como hacen los algoritmos de ordenación basados en comparaciones como QuickSort y MergeSort, RadixSort utiliza la información de los dígitos de cada número para ordenar el array. El algoritmo comienza ordenando los elementos por su dígito menos significativo y continúa ordenándolos por cada dígito en orden de menor a mayor. En cada pasada, el algoritmo utiliza un algoritmo de ordenación estable para ordenar los elementos basados en el dígito actual.

El coste asintótico de RadixSort depende del número de dígitos "d". En el peor caso, el número de dígitos es igual a log(k), donde "k" es el rango de valores posibles de los elementos. Por lo tanto, el coste asintótico de RadixSort es O(log(k) * (n + k)). Sin embargo, en la práctica, es más común utilizar una base de ordenación de 10, lo que significa que el número de dígitos es igual a log10(k). En este caso, el coste asintótico se reduce a O(log10(k) * (n + k)), que se puede simplificar a O(d * (n + k)).

En resumen, el coste asintótico de RadixSort es de O(d * (n + k)), lo que significa que el tiempo de ejecución del algoritmo aumenta proporcionalmente a d, n y k en el peor de los casos. Esto hace que RadixSort sea un algoritmo eficiente para ordenar grandes conjuntos de datos cuando el rango de valores es conocido y no es demasiado grande en comparación con el número de elementos, siempre y cuando los elementos se puedan categorizar en conjuntos, véase dígitos decimales de un número, caracteres alfabéticos en palabras, barajas de cartas…

C. QuickSort

El coste asintótico de ordenar un array de enteros mediante el algoritmo QuickSort es de O(n*log(n)), donde "n" es el número de elementos en el array.

El algoritmo QuickSort es un algoritmo de ordenación basado en la técnica de "dividir y conquistar". En cada iteración, el algoritmo selecciona un elemento del array, llamado "pivot", y divide el array en dos subarrays, uno con elementos menores que el pivot y otro con elementos mayores o iguales al pivot. Luego, el algoritmo aplica recursivamente la misma estrategia a los subarrays hasta que el array esté completamente ordenado.

El coste asintótico de QuickSort es de O(nlog(n)) porque el algoritmo divide el array en dos subarrays en cada iteración, lo que significa que el número de iteraciones es igual a log(n) en el peor caso. En cada iteración, el algoritmo realiza n comparaciones para dividir el array y n comparaciones para combinar los subarrays en un array ordenado, lo que da un coste total de nlog(n) comparaciones.

Sin embargo, en el peor caso, cuando el pivot elegido es el elemento más pequeño o el más grande del array, el algoritmo puede tener un coste asintótico de $O(n^2)$. Para evitar este problema, se pueden utilizar varias técnicas, como elegir el pivote de forma aleatoria o seleccionar como pivot el elemento correspondiente a la mediana de esa subdivisión.

D. SelectionSort

El coste asintótico de ordenar un array de enteros mediante el algoritmo SelectionSort es de $O(n^2)$, donde "n" es el número de elementos en el array.

El algoritmo SelectionSort es un algoritmo de ordenación simple que funciona encontrando repetidamente el elemento más pequeño del array y colocándolo en su posición correcta en el array ordenado. En cada iteración, el algoritmo busca el elemento más pequeño en el subarray no ordenado y lo intercambia con el elemento en la posición actual.

El algoritmo SelectionSort tiene un coste asintótico de tiempo de $O(n^2)$ porque necesita comparar cada par de elementos en el array (n*(n-1)/2 comparaciones) y hacer n intercambios. Esto hace que el algoritmo sea ineficiente para ordenar grandes conjuntos de datos. Sin embargo, el algoritmo puede ser útil para ordenar pequeños conjuntos de datos o para implementaciones simples en sistemas con recursos limitados.

En resumen, el coste asintótico de ordenar un array de enteros mediante el algoritmo SelectionSort es de $O(n^2)$, lo que significa que el tiempo de ejecución del algoritmo aumenta cuadráticamente con el tamaño del array.

E. HashMap

El coste asintótico de obtener la moda de un vector de enteros mediante un hashmap es de O(n), donde "n" es el número de elementos en el vector.

Para obtener la moda de un vector de enteros utilizando un hashmap, realizamos una iteración por el vector y guardamos en el hashmap el número de ocurrencias de cada elemento. Luego, iteramos por el hashmap para encontrar el elemento con el mayor número de ocurrencias, lo que corresponde a la moda.

En la primera iteración, se realiza una operación de tiempo

constante para insertar cada elemento en el hashmap, lo que resulta en un coste asintótico de O(n). En la segunda iteración, se realiza una operación de tiempo constante para buscar el elemento con el mayor número de ocurrencias en el hashmap, lo que también resulta en un coste asintótico de O(n).

Por lo tanto, el coste asintótico total de obtener la moda de un vector de enteros mediante un hashmap es de O(n), lo que significa que el tiempo de ejecución del algoritmo aumenta proporcionalmente al número de elementos en el vector. Esto hace que el algoritmo sea muy eficiente para conjuntos de datos grandes.

VII. Implementación del patrón MVC

Nuestra implementación del patrón Modelo-Vista-Controlador (MVC) es una combinación de dos enfoques. Por un lado, utilizamos un enfoque centralizado en el cual tenemos una interfaz implementada en el main que se encarga de recibir notificaciones de los componentes, como el controlador, para que desde el main se notifique a la vista de que inicie un proceso de pintado. Este enfoque se asemeja al MVC centralizado puro. Por otro lado, también utilizamos elementos del patrón por eventos, en el cual el controlador cuenta con un puntero del modelo, permitiendo que las comunicaciones se produzcan de forma directa entre dichos componentes con simples llamadas a métodos, sin la necesidad de pasar por el main. Debido a que hemos combinado características de ambos patrones según nuestras necesidades, no podemos definir nuestra implementación de forma específica a un solo tipo de MVC.

Al iniciar el programa, se inicia el modelo y la vista. El modelo no tiene puntero a ningún otro componente, ya que la comunicación con el modelo se realiza de forma unidireccional, desde el controlador o la vista hacia el modelo. Sin embargo, el controlador y la vista tienen punteros tanto al modelo como al main. Una vez inicializados los componentes, asignamos un valor de verdadero a una variable que se encarga de gestionar el pintado de las gráficas a través de la vista. Además, se inicia el controlador mediante una llamada al método start, el cual inicia un hilo de ejecución.

Dentro de este hilo de ejecución, el controlador se comunica con el modelo para utilizar los datos que se han creado de forma aleatoria y realizar los diferentes algoritmos con ellos. El controlador guarda los tiempos de ejecución de cada algoritmo y luego se comunica con el main para notificar a la vista que puede pintar los nuevos datos recogidos. Los datos son recogidos desde la vista, que mediante el puntero al modelo, hace una llamada a un método que le permite recoger los tiempos de ejecución obtenidos de cada uno de los algoritmos realizados desde el controlador. Una vez que se han recogido, la vista espera a que el controlador notifique al main de que todo está listo para actualizar la gráfica.

En resumen, nuestra implementación del patrón MVC es una combinación de enfoques centralizados y por eventos. Hemos utilizado diferentes características de ambos patrones según nuestras necesidades. Al iniciar el programa, el modelo y la vista se inician y el controlador se ejecuta en un hilo separado. El controlador utiliza los datos del modelo para realizar los algoritmos y guarda los tiempos de ejecución. Luego, el controlador se comunica con el main para notificar a la vista de que puede pintar los nuevos datos recogidos. La vista recoge los tiempos de ejecución y espera a que el controlador notifique al main para actualizar la gráfica.

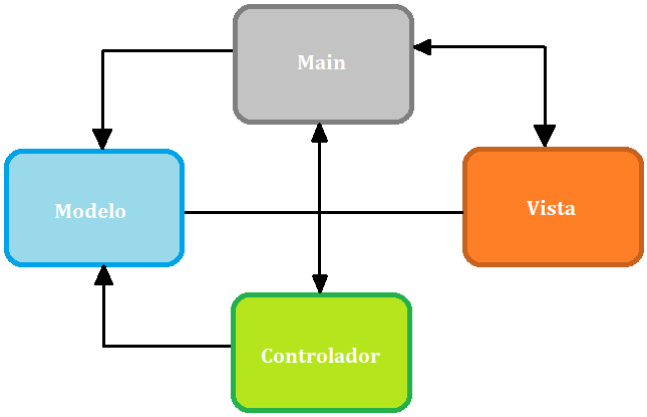

Fig. 2. Patrón MVC implementado en la práctica.

VIII. Implementación del Modelo

Para la implementación del modelo de datos hemos usado una sola clase principal:

1. Model.java: Contendrá todos los datos usados para el programa, usados por el Controlador y la Vista.

Model.java

Las variables principales de esta clase son:

- **elementos:** Array de N elementos (integers).
- **nElementos:** Integer que indica el tamaño del array de elementos.
- **tiempoActual:** Array usado para almacenar el tiempo de los tres algoritmos en la iteración actual
- **tiempos:** ArrayList de los tiempos que han tardado cada uno de los algoritmos en cada iteración. Es decir, ArrayList de "tiempoActual".
- **BufferSuavizado:** Buffer de tiempos para suavizar las medidas con la media de los tiempos posteriors y el actual.
- **SUAVIZADO:** Constante que indica la cantidad de medidas que se guardarán para realizar el suavizado.

Además de las variables, tendremos varios métodos y funciones que usarán otras clases (o la misma clase). Estos métodos y/o funciones son:

métodos privados:

- **rellenarElementos():** método encargado de rellenar el array de elementos con números aleatorios (1 a 500).

métodos públicos (usados por el controlador)

- **newIteration():** este método se encargará de aumentar el tamaño del array de elementos en una cantidad fija cada iteración (por defecto un incremento de 5000) y crea un nuevo array de elementos del nuevo tamaño N. En adición, reinicia la variable tiempoActual, para guardar los nuevos tiempos de los algoritmos en la iteración actual.
- **SetTiempo(i, t):** con este método podremos actualizar los valores de la variable "tiempoActual" y almacenar el tiempo (t) que tardó el algoritmo (i). Este i indica el indice en el array de tiempos.
- **SaveTiempo():** método encargado de guardar el tiempoActual de la iteración actual en el ArrayList de tiempos. En caso de que esté el suavizado activado (SUAVIZADO > 0), rellenaremos el buffer de suavizado y calcularemos la media entre la siguiente a pintar y las medidas guardadas en el buffer, luego introducimos el primer valor que llegó al buffer al ArrayList de tiempos e introducimos en el buffer el tiempoActual.
- **GetElementos():** función que nos devuelve una copia del array de elementos .
- **GetIteration():** función que nos devuelve el tamaño actual del array de elementos (N).

Métodos públicos (usados por la vista GraphPanel)

- **GetTiempo():** función que nos devuelve el último tiempo guardado en la lista de tiempos.

La utilidad principal de la clase modelo será, la creación del array de elementos, que será usado por el controlador. Una vez que se han enviado los elementos, el controlador realizará sus algoritmos programados para encontrar la moda y guardará cada uno de esos tiempos mediante la función **setTiempo(i,t)**.

Una vez que ya hayan ejecutado los 3 algoritmos y se hayan guardado. El controlador llamará al método **saveTiempo()**, para guardar los tiempos de la iteración actual. En adición, se llamará al método **newIteration()**, para aumentar el tamaño del array de elementos y generar un nuevo array de estos.

En cuanto a la Vista, la función **getTiempo()** será llamado por la clase **GraphPanel** cada vez que se quiera actualizar la gráfica de los tiempos.

En un primer lugar pensamos en la implementación del modelo utilizando un ArrayList distinto por cada algoritmo y luego, a la hora de guardar y obtener el último tiempo guardado sería más sencillo, pero tendríamos 2 desventajas:

1. Por un lado, ocuparíamos más memoria a medida que trascurre el programa y tiempo de ejecución dependiendo de la implementación de la lista enlazada por el Compilador.
2. En adición, también perderíamos flexibilidad a la hora de realizar cambios. Si quisiéramos representar 20 algoritmos tendríamos que crear 20 ArrayLists y sería más tedioso a la hora de guardar cada uno de los tiempos.

Por esa razón, decidimos utilizar un único ArrayList compuesto por arrays de Tiempos de ejecución. Por lo que, si en un momento quisiéramos guardar tiempos de ejecución de 20 algoritmos distintos, únicamente tendríamos que cambiar el tamaño del Tiempo actual (3 por defecto), y las funciones tanto de SetTiempo() y SaveTiempo, como de getTiempo() no se verían modificadas.

Una posible mejora sería guardar este valor 3 como una variable de la clase, la cual se actualiza en el constructor de la clase. Por lo que nos permitiría adaptarnos con facilidad a lo que requiera el controlador, que es uno de los propósitos de esta práctica: uso del MVC y ver sus ventajas y/o utilidades respecto a otro tipo de Arquitecturas.

Por último, me gustaría destacar que el suavizado, se realiza a la hora de guardar los tiempos de ejecución del algoritmo, y no durante la ejecución de este. Por lo que los tiempos no se ven afectados. Su función únicamente es para la visualización de estos y no haya picos muy altos o bajos durante la representación. Debido a que durante la ejecución del programa siempre habrá otros procesos que utilicen en segundo plano la CPU y nunca son exactos los tiempos.

IX. Implementación de la Vista

Para la implementación de la vista usaremos 2 clases principales

1. **View.java**: JFrame que contendrá todos los componentes de la interfaz y permitirá la interacción entre el usuario y la aplicación
2. **GraphPanel.java**: JPanel encargado de toda la visualización de los tiempos de ejecución de los algoritmos mediante una gráfica.

View.java

Esta clase será una extensión de un JFrame en el que estarán ubicados todos los componentes de la visualización y será el interfaz entre La vista y el programa Principal. Además, implementará la interfaz **ActionPerformed,** para la detección de eventos de los botones de la interfaz.

Las variables que tendrá esta clase son:

- **Height** y **Width:** indican el ancho y la altura del JFrame
- **StartB y StopB:** JButtons encargados de recibir el evento mandado por el ActionListener, serán usados por el usuario para empezar(StartB) el controlador y parar (StopB) la ejecución de este. Más adelante se explicará con detalle cómo se realizará la notificación al controlador.
- **Prog:** Puntero al Programa principal
- **margenLeyenda:** Constante para el posicionamiento y la distribución de los distintos componentes de la Vista.
- **Alg:** Listado de algoritmos (String)
- **algoritmos:** JComboBox usado para seleccionar que algoritmos se usará para el reescalado de la gráfica. El rescalado se explicará juntamente con la clase **GraphPanel**.
- **nElementos:** Clase Nelementos (Jpanel) que indicará el

tamaño del array en la iteración Actual del programa.
- **graphP:** Puntero a la clase GraphPanel.

Esta clase View.java tendrá 2 clases privadas únicamente accesibles por esta. Ambas clases serán JPanel.

- **Leyenda:** Mostrará una leyenda de los 3 algoritmos que usamos y su color correspondiente a la gráfica.
- **Nelementos:** Se encargará de mostrar el tamaño del array en la iteración actual del programa y de ir actualizándolo a medida que se ejecuta el programa mediante el método **setN(s)** de la clase.

Los métodos y funciones de la clase View.java son:

métodos públicos (usados por el Programa Principal):

- **mostrar():** método para hacer visible el JFrame y, a su vez, sus componentes.
- **SetN(s):** interfaz para llamar a la función **setN(s)** de la clase NElementos.
- **resetModel(m)** y **setRunning(b):** interfaces para llamar a las funciones de la clase **GraphPanel**.

métodos protegidos (usados por GraphPanel)

- **getSelectedAlg():** función que retorna el nombre del algoritmo seleccionado como opción del reescalado.

Métodos implementados (interfaz ActionPerformed)

- **actionPerformed(e):** método encargado de detectar los eventos de la interfaz y notificar al Programa Principal (StartB y StopB) o al GraphPanel (JComboBox algoritmos) dependiendo de la fuente del evento.

La creación del JFrame se formaría de la siguiente forma: En primer lugar, especificaríamos las dimensiones, el título y el fondo del JFrame. A continuación, iríamos creando todos los componentes que lo forman (Botones de Start y Stop, Seleccionador del algoritmo, Leyenda de los algoritmos, JPanel de los NElementos y, por último, el GraphPanel), asignando a los botones y el selector el ActionListener del JFrame para la detección de eventos de este.

Cabe destacar que la mayoría de los elementos están creados de forma escalar, es decir, las dimensiones y las posiciones entre los componentes del JFrame dependen de las dimensiones de este. Para que se pueda adaptar a la resolución del monitor en el que se lance el programa.

Por último, las clases privadas **NElementos y Leyenda**, se podrían haber diseñado sin necesidad de encapsularlas como clases, pero de la forma en la que se ha diseñado ha sido más fácil la especificación y el diseño de estos componentes para que quede de una forma más legible y entendible desde el punto de vista del programador.

GraphPanel.java

Esta clase será una extensión de un JPanel y será donde se pintará la gráfica con los tiempos de ejecución de cada algoritmo para el tamaño indicado del array

Las variables que tendrá esta clase son:

- **h** y **w:** indican el ancho y la altura del JPanel
- **view:** puntero a la clase View.java
- **mod:** puntero a la clase Model.java (Datos del Programa)
- **yValues, zValues, wValues:** ArrayList de los tiempos de ejecución de cada uno de los algoritmos.
- **Reescalar:** puntero a uno de los 3 arrayLists, servirá para reescalar la gráfica a partir del algoritmo que apunte.
- **startX, startY, endX, endY:** cordenadas de inicio y fin del rectángulo asociado al gráfico
- **unitX, unitY:** Relación entre pixeles y las unidades de ambos ejes.
- **PrevX, prevY0, prevY1, prevY2:** Variables que contendrán los valores previos tanto del eje X como del eje Y de las representaciones de los tiempos de ejecución del algoritmo.
- **Margin:** constante para el margen de los bordes del jPanel y el gráfico
- **running:** variable booleana que controla la ejecución del paintComponent y evita que se acceda al modelo mientras se está creando el JPanel.
- **changingRescaling:** variable booleana que controla la ejecución del paintComponent y evita que se pinten puntos extra al cambiar el algoritmo de reescalado.

Este JPanel será una gráfica donde el eje de la X indicará el Tamaño del array de Elementos (N) y el eje de la Y indicará el Tiempo de ejecución del algoritmo. Cada línea vendrá pintada de un color distinto y corresponderá a un algoritmo, indicado en la Leyenda.

Los métodos y funciones de la clase **GraphPanel** son:

Métodos privados:

- **init():** método encargado de inicializar las listas de tiempos, preparar el array de reescalado y reiniciar las variables de relacionadas con la gráfica.
- **AddPoint():** método encargado de añadir los últimos tiempos de cada algoritmo a su ArrayList correspondiente

Métodos públicos (usados por la clase View):

- **setReescalarY():** método encargado de apuntar al arrayList yValues para usarla como referencia de escalado.
- **setReescalarZ():** método encargado de apuntar al arrayList zValues para usarla como referencia de escalado.
- **setReescalarW():**método encargado de apuntar al arrayList wValues para usarla como referencia de escalado.

- **ChangingRescaling()**: setter de la variable "changingRescaling" a true;
- **setRunning(b):** setter de la variable running con el valor 'b'.
- **resetModel(m):** apunta a un nuevo modelo de Datos y llama a la función **init()** ya explicada.

Métodos sobrescritos de la clase JPanel

- **repaint():** función de chequeo para asegurar que nuestra variable Graphics (JPanel) está inicializada.
- **paintComponent(g)**: Método encargado de pintar el Jpanel.

En primer lugar, comprobamos que nuestro Modelo existe y estamos inicializados (Running = true). Si es así, continuaremos con el pintado.

En segundo lugar, recorreremos la lista apuntada por la variable "reescalar" y cogeremos y calcularemos los datos necesarios para el dibujo de los tiempos de ejecución.

A continuación, recorreremos cada una de las listas de tiempos e iremos pintando las líneas de la gráfica, ponderando con los valores obtenidos en el paso anterior

Por último, dibujaremos los ejes X e Y, y todos sus valores, en forma matricial haciendo uso de las variables unitX y unitY para saber las coordenadas donde pintar los valores de los ejes y las divisiones de los ejes. También añadimos los títulos de los ejes y de la gráfica.

Cabe destacar que esta clase se podría mejorar para adaptarla de manera similar al modelo, utilizando un solo ArrayList con los tiempos en vez de un ArrayList por cada algoritmo, de manera que sería más fácil adaptar (por ejemplo) la visualización de 3 a 20 algoritmos al igual que se ha explicado antes. Y fusionar los métodos de **setReescalarY(), setReescalarZ(), setReescalarW()** en un solo método **setReescalar(i)**, en el que se indica mediante el índice "i" cuál de los algoritmos se usaría para el reescalado. Se tendrá en cuenta en caso de realizar mejoras posteriormente.

X. Implementación del Controlador

En el paquete de controlador de nuestra práctica disponemos de dos clases. La primera es la clase Sorting.java donde hemos implementado todos los algoritmos de ordenación mencionados en el apartado VI. Algoritmos implementados. La segunda clase es Controller.java, donde hemos implementado el controlador de nuestro patrón MVC.

La clase controlador extiende la clase **Thread** de java para permitirnos realizar simultáneamente los cálculos necesarios en el controlador y la visualización por pantalla e interacción con el usuario en la vista.

En el controlador disponemos de los tres algoritmos que se ejecutan en secuencia, midiendo los tiempos de ejecución de cada uno de ellos para después mostrarlo en la gráfica.

El hilo de ejecución se basa en iteraciones. En cada una de las iteraciones llamamos al método **newIteration()** mediante el puntero al modelo del que disponemos en la clase. En este método, genera aleatoriamente un array de enteros de tamaño n. El tamaño del array comienza con 100 elementos y se incrementa en 5000 en cada iteración del bucle presente en el hilo. Dentro de este bucle, guardamos los nanosegundos del sistema en una variable, ejecutamos el primer algoritmo para encontrar la moda del array de enteros y cuando acaba, hacemos la diferencia de los tiempos para saber el tiempo de respuesta de ese algoritmo y lo guardamos en una array de elementos de tipo ***long*** llamado **tiempoActual** mediante **model.setTiempo()** (El primer parámetro es la posición donde se guarda el tiempo de respuesta de cada algoritmo y el segundo es el propio tiempo de respuesta). El hilo repite el mismo proceso para los dos siguientes algoritmos y al acabar con el último, hace una llamada a **saveTiempo()** que almacena los tiempos de respuesta de los tres algoritmos guardados en **tiempoActual** en el ArrayList tiempos, que contendrá los puntos que la vista graficará. Finalmente notificará al programa para que mande a la vista que dibuje los nuevos puntos que ha introducido.
El hilo seguirá su ejecución hasta que se presione el botón STOP, que llamará desde la vista al método **parar()** del controlador, establecerá la variable de clase seguir a false y al acabar los cálculos de la iteración actual, terminará el bucle y el proceso finalizará.

En el caso del producto vectorial, al tener un coste asintótico de $O(n^2)$, rápidamente se convierte en el cuello de botella del hilo. Por lo tanto, hemos decidido que a partir de un tamaño del array indicado en **MAX_PRODUCT_VECT**, el hilo dejará de ejecutar ese algoritmo, ya que los tiempos de respuesta a partir de ese tamaño de n, no se pueden apreciar en la gráfica al mismo tiempo que se puedan apreciar los tiempos de respuesta de los otros dos algoritmos. El tamaño a partir del cual hemos decidido que deje de calcular es 70.000 elementos (con ese tamaño suele tener un tiempo de respuesta de entre 1 y 2 segundos).

Además del hilo del controlador, contamos con los 3 algoritmos cuyo rendimiento se pone a prueba:

A. SortAndGet

El método **sortAndGet()** utiliza uno de los métodos de ordenación que hemos definido en la clase Sorting.java y después hace un recorrido sobre el array de enteros ordenado para encontrar la moda. Para llevar a cabo esta tarea, lo que hacemos es tener contabilizar la cantidad de veces que aparece el mismo número en el array hasta que aparece un elemento distinto. Cada vez que encontramos el mismo elemento, aumentamos la cantidad de veces que aparece y si su frecuencia es mayor que la máxima encontrada, ese elemento se almacena como la moda del vector. Cuando acaba el algoritmo, la moda es el elemento con mayor frecuencia de aparición.

Para la realización de la práctica hemos optado por utilizar el método de ordenación **mergeSort()** para poder apreciar en la gráfica los tiempos de respuesta relativos entre un algoritmo $O(n^2)$, $O(n \cdot \log(n))$ y $O(n)$. En este caso **sortAndGet()** representaría ese algoritmo $O(n \cdot \log(n))$ ya que la ordenación tiene un coste $O(n \cdot \log(n))$ y la búsqueda de la máxima frecuencia sería $O(n)$, lo que implica que **sortAndGet()** sería $O(2n \cdot \log(n))$, resumiendo: $O(n \cdot \log(n))$.

Este modo de encontrar la moda sería aproximadamente igual de eficiente que **hashing()** si utilizásemos el método **radixSort()**, que al igual que **hashing()**, se ve favorecido por enteros con pocos dígitos, que se traducen en menos iteraciones del algoritmo **countSort()** utilizado por este.

B. Hashing

El método **hashing()**, al igual que **sortAndGet()**, trata de encontrar la moda en un vector de enteros. Sin embargo, a diferencia de **sortAndGet()**, **hashing()** utiliza una tabla de Hash en lugar de ordenar los elementos. Como bien sabemos, la inserción de un elemento en una tabla Hash tiene un costo de entre O(1) y O(n·log(n)). Por lo tanto, introducir todos los elementos tiene un coste de entre O(n) y O(n·log(n)). A la hora de introducir los elementos en la tabla, utilizamos el entero como la clave, y el valor asociado a la clave es la cantidad de veces que se ha tratado de introducir el elemento. Es decir, antes de introducir el elemento, busca si ya existe en la tabla Hash, y de ser el caso, incrementa en uno el valor asociado a esa clave. De lo contrario, si el elemento no está presente, se introduce con un valor inicial de 1.

Una vez que se han procesado todos los elementos del array, hacemos un recorrido sobre la tabla Hash para encontrar el valor más alto, es decir, el elemento que aparece más veces en el array.

Como hemos mencionado en el apartado anterior, encontrar la moda con este algoritmo es especialmente rápido cuando el rango de números es pequeño, de este modo el segundo recorrido sobre la tabla será más corto al haber menos parejas clave-valor en la tabla.

C. Producto vectorial

En este algoritmo realizamos la multiplicación de un vector de enteros por sí mismo, siendo el propio vector una matriz columna y una matriz fila en los operandos de la multiplicación de matrices. Dicho de otra forma, realizamos la multiplicación del primer elemento del array por cada uno de los elementos del propio array y sumamos los resultados, almacenándolos en la primera posición del array de la solución. A continuación, realizamos la multiplicación del segundo elemento por todos los elementos y así sucesivamente hasta llegar al último. En pocas palabras, se hace un recorrido anidado en el recorrido general del array, lo que supone un coste asintótico de $O(n^2)$.

XI. Suavizado

A la hora de tomar mediciones sobre el tiempo de respuesta de los distintos algoritmos, hemos tenido que recurrir al tiempo del sistema en nanosegundos. Un problema que representa esta práctica es el ruido que introducen el resto de funciones del sistema operativo con los que se intercala el tiempo de CPU, que es impredecible.

Para estabilizar las medidas y evitar los picos incontrolables en las medidas causadas por el intercalado de procesos, hemos decidido hacer un suavizado de las medidas como ha quedado explicado en el apartado VIII. Implementación del Modelo, utilizando un buffer para almacenar SUAVIZADO medidas antes de realizar las medias entre ellas y pintar el eje siguiente punto.

A la hora de elegir la cantidad de medidas con las que se hace la media, debemos elegir una cantidad adecuada, es decir, suficientes como para que se realice el suavizado y, al mismo tiempo, no demasiado grande como para que afecte significativamente a las medidas representadas en la gráfica.

Para encontrar este valor de SUAVIZADO, hemos realizado en pruebas con distintos valores, anotando el tiempo de respuesta de los algoritmos de **sortAndGet** y **hashing** para un tamaño del array de 2.560.100 elementos.

	Tiempos de respuesta (ms)	
SUAVIZADO	**sortAndGet**	**hashing**
0	280	50
2	280	50
5	280	50
10	250	40

Tabla 1. Medidas del tiempo de respuesta de los algoritmos en función del suavizado para una n = 2.560.100.

Como se puede apreciar en la Tabla 1, con un SUAVIZADO mayor que 5, los valores del tiempo de respuesta comienzan a distorsionarse, por lo tanto, un SUAVIZADO de 5 evita los picos pronunciados al mismo tiempo que mantiene la coherencia de los datos.

XII. Reflexión sobre el rendimiento de los algoritmos

A la hora de analizar el rendimiento de los métodos de sortAndGet y hashing, cabe mencionar que la eficiencia en espacio y tiempo puede depender de los siguientes factores:

1. **Algoritmo de ordenación empleado en sortAndGet**: En el algoritmo sortAndGet, tenemos dos etapas, la primera etapa de ordenación del array, y la segunda de búsqueda de la mayor frecuencia. Independientemente del algoritmo de ordenación utilizado en la primera etapa, el segundo recorrido para buscar la frecuencia siempre tiene un coste de O(n). Por lo tanto, si utilizamos **selectionSort()** para ordenar, el coste general del algoritmo sería $O(n^2)$. Por defecto, nuestro proyecto tiene seleccionado como método de ordenación el método **mergeSort()**, con coste O(n·log(n)). Sin embargo, si utilizáramos el método **radixSort()**, que tiene un coste de O(n·m), sortAndGet tendría un coste asintótico O(2n·m), que en resumidas cuentas sería O(n) ya que la m sería la cantidad de dígitos en base 10 del entero más grande, que de utilizar **Integer.MAX_NUMBER** como máximo, significaría un coste O(20·n) que seguiría siendo O(n), no obstante, dependerá del siguiente factor.

2. **Rango de valores de los enteros del array**: en nuestra práctica, hemos seleccionado un rango de valores para los enteros del array de entre 1 y 500. Este rango de

valores es bastante pequeño, lo que favorece al algoritmo de hashing, y al algoritmo sortAndGet si hubiéramos utilizado **radixSort()** al realizar la ordenación. El algoritmo de hashing funciona mejor con un rango pequeño de valores, ya que tendrá como máximo 500 claves en la tabla, mientras que si utilizamos hasta **Integer.MAX_NUMBER** claves, la tabla de Hash tendrá que reorganizar la memoria frecuentemente, además de ocupar mucho más espacio. Del mismo modo, **radixSort()** realiza diversas pasadas por el array de entrada, ordenando primero por el último entero, después por el penúltimo, y así sucesivamente hasta haber hecho tantas pasadas como cantidad de dígitos en base 10 tenga el mayor elemento, que de ser **Integer.MAX_NUMBER**, serían 10 pasadas (2^{31} - 1 = 2.147.483.647). En cuanto al coste en espacio de **radixSort()**, crece linealmente con respecto al tamaño del array ya que depende de la base utilizada y el propio tamaño el array, es decir, O(10*n).

Para salir de dudas sobre qué algoritmo sería más eficiente para distintos tamaños del array y para distintos algoritmos de ejecución, hemos realizado pruebas y estos han sido los resultados en cada caso:

- Enteros grandes y **mergeSort()**: sortAndGet y hashing tienen un tiempo de respuesta prácticamente idénticos, pero sortAndGet es ligeramente más rápido.
- Enteros grandes y **radixSort()**: sortAndGet es considerablemente más rápido.
- Enteros pequeños y **mergeSort()**: hashing es mucho más eficiente.
- Enteros pequeños y **radixSort()**: sortAndGet y hashing tienen un tiempo de respuesta prácticamente idénticos, pero hashing es ligeramente más rápido.

Concluyendo, para rangos de enteros pequeños (500 valores) siempre utilizaría el método de hashing para buscar la moda. Para rangos de enteros grandes (2.147.483.647 valores) utilizaría sortAndGet con el método **radixSort()** siempre y cuando la memoria sea suficiente, si tuviéramos limitación de memoria, utilizaría **mergeSort()**.

XIII. Vídeo Demostrativo

En la cabecera de cada clase del proyecto NetBeans, encontraremos un enlace a un vídeo de YouTube donde se muestra la estructura del proyecto, las partes más relevantes del código fuente y el funcionamiento de la aplicación con su interfaz gráfica.

XIV. Distribución del trabajo

En esta práctica han colaborado todos los miembros del equipo en similar medida, añadiendo cada uno funcionalidades a la aplicación a medida que han surgido nuevas ideas y funcionalidades.

XV. Guía de usuario

Para ejecutar el programa, compilaremos el proyecto de NetBeans y haremos un ***run project*** del archivo Practica1AlgAv.java.

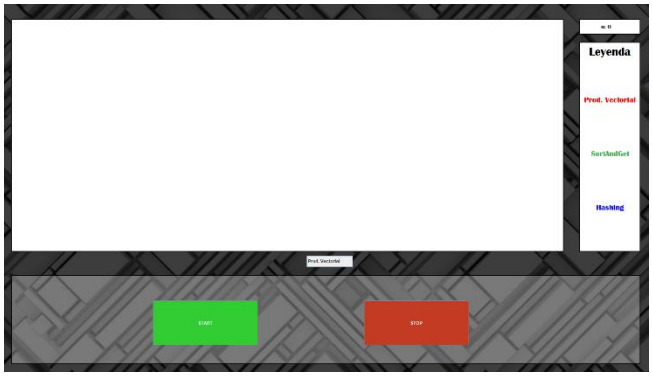

Fig. 3. Interfaz de usuario antes de la ejecución.

Acto seguido nos aparecerá la siguiente ventana que cuenta con los siguientes elementos:

- Un panel donde se dibujarán los gráficos.
- Una leyenda de los datos que se muestran, enseñando el color con el que se pintará el tiempo de respuesta en función de la n (tamaño del array) para cada algoritmo.
- Un contador para mostrar el tamaño del array con el que está ejecutando los algoritmos.
- Un menú desplegable en la parte inferior del panel de dibujo, que permite seleccionar los tres algoritmos. Al seleccionar uno de ellos, el eje Y que representa el tiempo de respuesta, se ajustará al tiempo de respuesta más alto que ha tenido ese algoritmo. Por defecto, el algoritmo seleccionado es el producto vectorial, que es el que mejor nos permite apreciar la relación entre los tres.
  En cualquier momento del programa podemos cambiar el algoritmo seleccionado.
- Un botón de STOP que detiene al controlador de seguir iterando, dejando que termine los cálculos de la iteración en la que se encuentra.
- Un botón de START para empezar a calcular los tiempos de respuesta de los algoritmos y a graficarlos a medida que va aumentando la n y, en consecuencia, los valores de los ejes.
  Una vez detenida la ejecución, podemos volver a reanudar los cálculos desde 0 pulsando este botón.

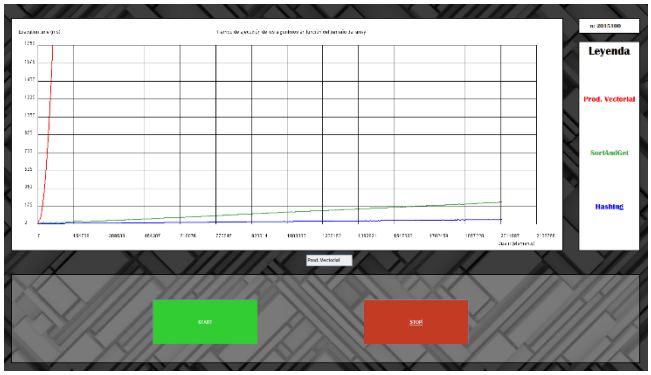

Fig. 4. Interfaz de usuario pausada tras el inicio (enfocando el algoritmo del producto vectorial).

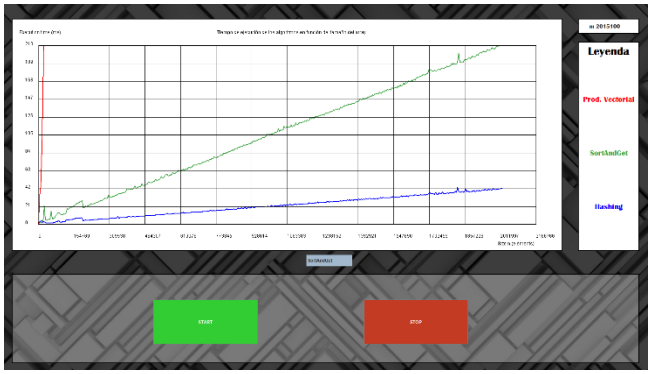

Fig. 5. Interfaz de usuario pausada tras el inicio (enfocando el algoritmo sortAndGet()).

XIII. Conclusión

Esta práctica nos ha permitido experimentar y realizar una implementación de un programa usando MVC y conocer la importancia del coste asintótico cuando se realiza un algoritmo.

También nos ha servido para recordar y asimilar conceptos de Swing en Java.

Agradecimiento

Gracias al Dr. Miguel Mascaró Portells en la supervisión, orientación constante y apoyo en todo el proyecto.

Referencias

BRASSARD, G; BRATLEY, P., Fundamentals of Algorithmics, Prentice Hall, 1995.

KLEINBERG, J.; TARDOS, E. , Algorithm Design, Addison-Wesley, 2005.

CORMEN, T.; LEISERSON, C.; RIVEST; R.; STEIN, C., Introduction to Algorithms, The MIT Press, 2009.

SEDGEWICK, R. , Algorithms in C++: Part 1-4 & Part 5 (3rd ed), Addison-Wesley, 2002.

DASGUPTA, S.; PAPADIMITRIOU, C.; VAZIRANI, U. , Algorithms, McGraw-Hill, 2008.