Divide&Conquer

Martin Mitrovski Delov, Josep Antoni Naranjo Llompart, Sergi Moreno Pérez, Pau Rosado Muñoz

Resumen — El ejercicio consiste en desarrollar un programa capaz de calcular las tres parejas de puntos con menor distancia, de entre una nube de puntos 2D en un espacio equiprobable. El programa debe ser desarrollado bajo diversas condiciones y para diferentes algoritmos.

Por motivos de tamaño del fichero de la práctica, hemos decidido subir el vídeo a YouTube y no incluirlo en el archivo comprimido entregado. El enlace al vídeo de la práctica en YouTube es el siguiente:
https://youtu.be/xSB57Oy3RSg

I. INTRODUCCIÓN

En este artículo se explicará cómo se ha realizado la tercera práctica de la asignatura Algoritmos Avanzados. Se desarrollarán algunos conceptos teóricos expuestos en el tercer tema y también se describirá la implementación de una aplicación que calculará dentro de una nube de puntos cuáles son las tres parejas de puntos cuya distancia es menor.

Los puntos por tratar son los siguientes: el entorno de programación utilizado, los conceptos de MVC, el patrón por eventos, la técnica D&C y la implementación del MVC para el desarrollo de nuestra aplicación.

Por parte del concepto MVC, explicaremos qué es y por qué es el método de diseño que utilizaremos durante todo el curso para la implementación de las prácticas.

Explicaremos en detalle la técnica D&C, tratada en este tercer tema, sus características, ventajas, desventajas, etc.

Una vez explicadas estas secciones teóricas, procederemos a mostrar la implementación de la aplicación usando el patrón de diseño MVC y aplicando la técnica D&C para el cálculo. Explicaremos cada una de las partes, junto a los conceptos de código más importantes. Haremos bastante hincapié en el nivel de abstracción que se ha perseguido obtener para la implementación y comunicación entre los diferentes componentes de la arquitectura. Primero explicaremos el modelo de datos, después el controlador y finalmente la vista o interfaz gráfica.

Del apartado gráfico, explicaremos algunas características para asegurar la consistencia del programa.

Para acabar habrá un apartado de conclusiones dónde se realizará una breve reflexión sobre los resultados obtenidos y sobre el trabajo dedicado al proyecto.

II. ENTORNO DE PROGRAMACIÓN

En nuestra práctica usaremos el entorno de programación NetBeans, entorno de desarrollo integrado libre, orientado principalmente al desarrollo de aplicaciones Java. La plataforma NetBeans permite el desarrollo de aplicaciones estructuradas mediante un conjunto de componentes denominados “módulos”. La construcción de aplicaciones a partir de módulos las permite ser extendidas agregándoles nuevos módulos. Como los módulos pueden ser desarrollados independientemente, las aplicaciones implementadas en NetBeans pueden ser extendidas fácilmente por otros desarrolladores de *software*.

III. MODELO VISTA CONTROLADOR (MVC)

El modelo-vista-controlador (MVC) [1] es una arquitectura de software que separa los datos, la interfaz de usuario y la lógica de control en tres componentes distintos. El modelo representa los datos, la vista muestra los datos al usuario de forma visualmente atractiva y el controlador actúa como intermediario entre el modelo y la vista, gestionando el flujo de información entre ellos. El flujo de una aplicación que usa el MVC comienza con la interacción del usuario con la interfaz de usuario, el controlador recibe la acción del usuario y actualiza el modelo acorde a la acción, luego la vista utiliza los datos del modelo para generar la interfaz adecuada para el usuario.

Existen tres formas básicas de implementación del MVC: distribuido, centralizado y centralizado en el controlador. El uso de este patrón de diseño por capas tiene varias ventajas, como la división del trabajo, la reutilización de código, la flexibilidad y escalabilidad que supone, así como la facilidad de realizar el *testing* de la aplicación. Sin embargo, también tiene algunas desventajas, como la complejidad adicional que puede presentar debido a que requiere más código y esfuerzo para mantener la separación de tareas entre los componentes.

En la Figura 1 aparece el esquema habitual del patrón de diseño MVC.

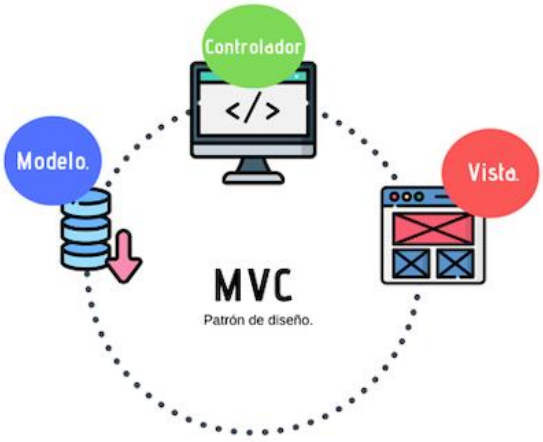

Figura 1: Modelo Vista Controlador

IV. PATRÓN POR EVENTOS

La arquitectura orientada a eventos o *Event-driven architecture* (EDA) es una forma de diseñar software que se enfoca en los eventos que ocurren dentro del sistema [2]. Los eventos son ocurrencias notificadas, como cuando un usuario hace clic en un botón. En una arquitectura EDA, los componentes del sistema están diseñados para reaccionar a estos eventos en tiempo real. Los componentes se comunican entre sí mediante eventos, propagándose a los componentes relevantes que responderán al evento adecuadamente.

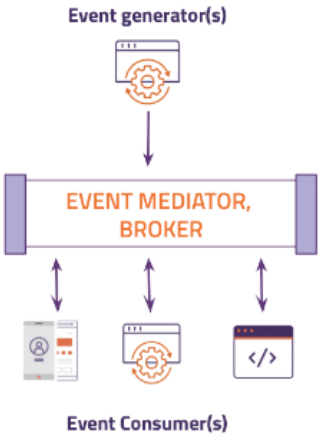

Figura 2: Patrón por eventos

La arquitectura de eventos es especialmente útil [3] en sistemas que requieren procesamiento en tiempo real, escalabilidad y flexibilidad, como redes de sensores, comercio financiero, sitios web de comercio electrónico y redes sociales. Los sistemas basados en esta arquitectura pueden reaccionar rápidamente a cambios y son tolerantes a fallos debido al bajo acoplamiento entre los componentes.

Sin embargo, esta arquitectura también puede tener desventajas, como la complejidad y dificultad de depuración debido a la naturaleza distribuida del sistema y la necesidad de sincronización de los eventos. También se deben tener en cuenta posibles problemas de latencia, ya que los eventos pueden tardar un cierto tiempo en propagarse a través de los componentes.

V. DIVIDE&CONQUER

El *Divide and conquer* (dividir y vencerás) [4] es un enfoque de resolución de problemas que se utiliza en muchos campos, como la ciencia de la computación, la matemática, la economía, la política, estrategias militares y otros.

El enfoque consiste en dividir un problema grande y complejo en subproblemas más pequeños y manejables que sean más fáciles de resolver. Luego, cada subproblema se resuelve por separado y los resultados se combinan para obtener una solución al problema original.

Por lo general, se divide el problema en subproblemas del mismo tipo, que se resuelven de manera recursiva hasta que los subproblemas sean lo suficientemente pequeños como para ser resueltos de manera directa. La recursividad es una técnica común utilizada en el enfoque "divide and conquer".

Este enfoque se utiliza en muchos algoritmos y técnicas de programación. Un ejemplo común es el algoritmo de ordenamiento "merge sort", que divide una lista de elementos en dos mitades, ordena cada mitad por separado y luego combina las dos mitades ordenadas en una sola lista ordenada. Este proceso de división y combinación se repite recursivamente hasta que la lista completa esté ordenada.

Otro ejemplo es el algoritmo "quicksort", que también se basa en el enfoque "divide and conquer". En este algoritmo, se elige un elemento "pivot" de la lista y se divide la lista en dos subconjuntos, uno que contenga todos los elementos menores que el pivote y otro que contenga todos los elementos mayores que el pivote. Luego, se ordenan los dos subconjuntos recursivamente utilizando el mismo algoritmo y se combinan en una sola lista ordenada.

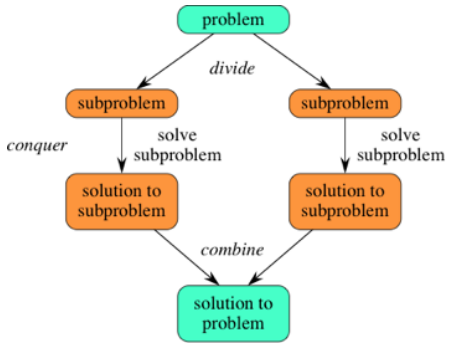

Figura 3: Divide and Conquer

V.I RECURSIVIDAD

Como se ha dicho en el apartado anterior, la recursividad es una técnica común utilizada en el enfoque de "divide and conquer", por lo que ahora se explicará este concepto:

La recursividad [5] se refiere a una función o proceso que se llama a sí misma repetidamente en su ejecución. Es una técnica común de la programación la cual se utiliza para resolver problemas que pueden ser descompuestos en problemas más pequeños (subproblema) con lo cual, en lugar de abordar el problema original directamente, resolvemos cada subproblema recursivamente.

Es necesario establecer un caso base que permita a la función finalizar la recursión y evitar caer en una llamada infinita.

Un ejemplo fácil para entender mejor la recursividad sería el cálculo factorial de un número entero.

La factorial de un número entero n (se denota como: n!), es el producto de todos los números enteros positivos desde 1 hasta n. Es decir:

n! = 1 x 2 x 3 x … x n

Por ejemplo, si n = 5, su factorial sería:

5! = 1 x 2 x 3 x 4 x 5 = 120

Para calcular la factorial de 5 utilizando recursividad, podemos definir una función recursiva que se llame a sí misma para calcular la factorial de n-1. La función tendría un caso base para cuando n=1, en cuyo caso la factorial sería simplemente 1.

Si llamamos a esta función con un argumento de 5, se llamará a sí misma con un argumento de 4, luego con 3, luego con 2, y finalmente con 1 (el caso base). La función comenzará a regresar valores una vez que alcance el caso base, y se multiplicará cada valor de retorno por n para obtener el factorial completo. La figura 4 ilustra todo esto explicado.

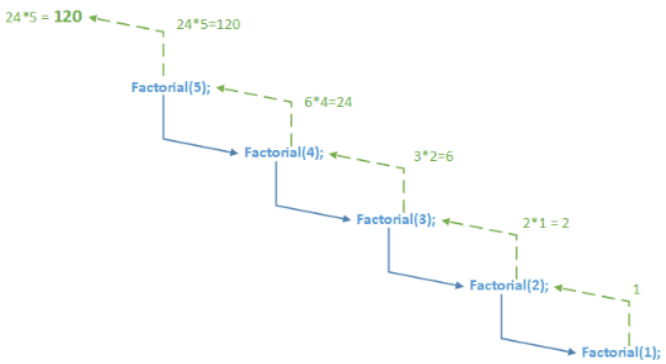

Figura 4: Cálculo de la factorial de un número entero

V.II ESQUEMA GENERAL

El esquema general de "divide and conquer"

1. Dividir: Descomponemos el problema en subproblemas del mismo tipo y los cuales son más manejables. Por lo general representa un enfoque recursivo para dividir el problema hasta que no sea posible crear más subproblemas y sean lo suficientemente pequeños como para ser resueltos de manera directa.
2. Vencer: Resolveremos los subproblemas de manera separada recursivamente. Este paso recibe un gran conjunto de subproblemas a ser resueltos y suelen resolverse de manera directa.
3. Combinar: Combinamos las respuestas apropiadamente. Cuando los subproblemas son resueltos, este paso los combina recursivamente hasta que estos formulen la solución al problema original. Esto puede implicar la unión de listas, la suma de números o cualquier otra operación necesaria para obtener la solución.

V.III CUANDO USAR EL *DIVIDE AND CONQUER* ¿SE PUEDE USAR SIEMPRE?

El enfoque "divide and conquer" se puede utilizar cuando se enfrentan problemas grandes y complejos que pueden ser divididos en subproblemas más pequeños y manejables. Es particularmente útil cuando estos subproblemas son del mismo tipo y pueden ser resueltos de manera recursiva.

Algunos ejemplos comunes de problemas que se pueden resolver utilizando "divide and conquer" son:

- Algoritmos de ordenamiento: como el Merge Sort o el Quick Sort, donde se divide una lista en subconjuntos más pequeños, se ordenan de manera recursiva y luego se combinan.
- Búsqueda de datos: como el Binary Search, donde se divide una lista en dos mitades y se busca en la mitad adecuada de manera recursiva.
- Multiplicación de matrices: donde se divide una matriz grande en submatrices más pequeñas, se multiplican de manera recursiva y luego se combinan.
- Cálculo de exponenciación: donde se divide un problema de exponenciación grande en problemas más pequeños y se resuelve de manera recursiva.
- Problemas de geometría computacional: como el cálculo de la envolvente convexa, donde se divide el

problema en subproblemas más pequeños y se resuelven de manera recursiva.

Respecto a la pregunta de si se puede usar siempre, debe cumplir todos estos requisitos:

- Se requiere un método más o menos directo para resolver los problemas pequeños (casos base).
- Se debe poder dividir en problemas más simples.
- Los subproblemas deben ser disjuntos (soluciones independientes).
- Deben de poderse combinar las sub-soluciones para poder obtener la solución mayor con un coste relativamente bajo (menor que el global).

V.IV ANALISIS DEL COSTE

El análisis del coste de "divide and conquer" [6] depende en gran medida de la complejidad de los subproblemas que se generan durante la división y del número de subproblemas que se crean.

El algoritmo "divide and conquer" se puede evaluar utilizando el segundo teorema de reducción si los subproblemas son de tamaño similar. La fórmula recursiva es T2(n) = k·T2(n/c) + coste (MAX(FRAGMENTAR, COMBINAR)). Sin embargo, esto no garantiza la eficiencia del algoritmo, ya que puede ser mejor, peor o igual que un algoritmo iterativo conocido.

La búsqueda dicotómica y la ordenación rápida, Quicksort, son aplicaciones clásicas de "divide and conquer". La búsqueda dicotómica tiene una complejidad de O(log n), mientras que Quicksort presenta un comportamiento menos homogéneo debido a la calidad del pivote utilizado. Con un buen pivote, Quicksort tiene una complejidad de O(n · log n), mientras que con un mal pivote la complejidad puede ser de O(n^2). A pesar de la complejidad en el peor caso, el Quicksort es un algoritmo de ordenación eficiente debido a su complejidad en el caso medio de O(n · log n).

V.V MERGE-SORT

El algoritmo de Merge-sort [7] es como hemos mencionado un algoritmo característico de ordenación basado en el paradigma "divide y vencerás". Su idea fundamental es la de dividir el array original en dos mitades iguales, ordenar cada mitad recursivamente con Merge-sort, y luego combinar ambas mitades ordenadas en un solo array ordenado.

El proceso de dividir se realiza hasta que se alcanzan arrays de un solo elemento, los cuales ya están ordenados por definición. Luego, el proceso de combinación de ambas mitades ordenadas se realiza a través de la función *merge*, que toma dos arrays ordenados y los combina en un solo array ordenado.

La implementación de la función *merge* consiste en comparar el primer elemento de cada array ordenado y colocar el menor en el array resultado, avanzando en el array correspondiente. Este proceso se repite hasta que se han recorrido todos los elementos de ambos arrays, y se añaden los elementos restantes del array que no se ha recorrido por completo.

En cuanto al tema coste, el algoritmo de Merge-sort tiene una complejidad temporal de O(n log n) en el peor y caso, lo que lo hace muy eficiente para ordenar grandes cantidades de datos.

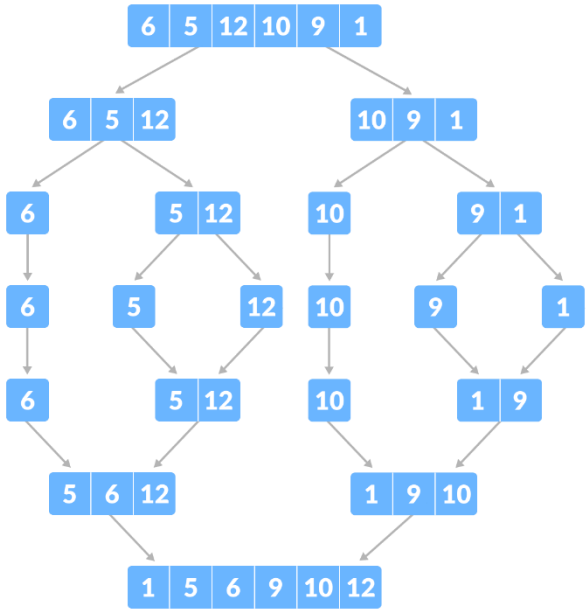

Figura 5: MERGE-SORT

En la figura 5 podemos apreciar como es el funcionamiento del merge-sort para un array de 6 elementos.

V.VI QUICKSORT

El algoritmo de Quicksort [8], como ya se ha mencionado, se trata de un algoritmo de ordenación basado en el paradigma "divide y vencerás", que se caracteriza por ser muy eficiente en la práctica. Su idea fundamental es la de elegir un elemento del array, llamado pivote, y dividir el array en dos sub-arrays: uno con los elementos menores al pivote y otro con los elementos mayores. Luego, el proceso se repite recursivamente para cada sub-array hasta que los arrays resultantes tengan un solo elemento, los cuales ya están ordenados por definición. Esto se puede ver reflejado en la figura 6.

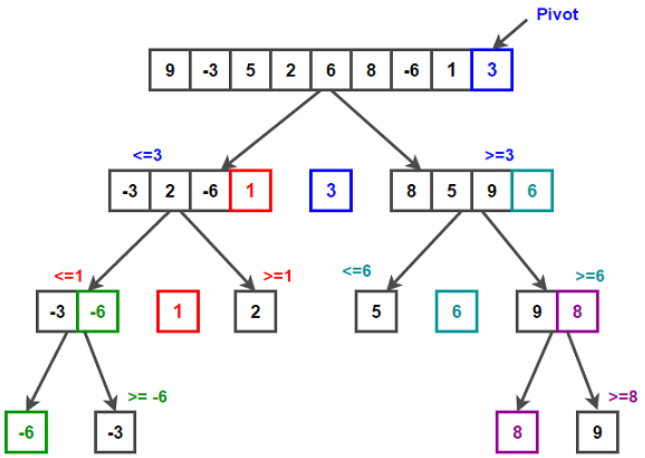

Figura 6: QUICKSORT

En la implementación del algoritmo de Quicksort, la elección del pivote es muy importante para garantizar una buena eficiencia en tiempo de ejecución. Una estrategia común es elegir el primer elemento del array como pivote, aunque esto puede generar un peor rendimiento en casos de arrays ordenados o casi ordenados. Otra estrategia es elegir un pivote aleatorio, lo cual mejora la eficiencia promedio del algoritmo.

Una vez elegido el pivote, el proceso de partición consiste en comparar cada elemento del array con el pivote y colocarlos en la parte correspondiente del array resultante. Este proceso se realiza de tal forma que todos los elementos a la izquierda del pivote son menores o iguales que el pivote, y todos los elementos a la derecha son mayores o iguales. Luego, se repite el proceso de Quicksort recursivamente para cada sub-array generado por la partición.

Por lo que respecta al coste, este algoritmo tiene una complejidad temporal promedio de O(n·log n), lo que lo hace muy eficiente para ordenar grandes cantidades de datos. Sin embargo, su complejidad en el peor caso puede llegar a ser O(n^2) si se elige un pivote inadecuado, lo que puede ocurrir en casos específicos de datos ya ordenados o casi ordenados.

VI. IMPLEMENTACIÓN MVC

Como ya hemos comentado anteriormente en este informe existen tres tipos de variaciones según respecta a la implementación del patrón “Modelo Vista Controlador”. En esta práctica, debido a la facilidad que ofrece a la hora de gestionar los distintos módulos, así como para facilitar la compresión de los distintos componentes sin mezclar funcionalidades, se ha optado por desarrollar el programa siguiendo un esquema centralizado.

Para refrescar cabe comentar que este esquema persigue que la gestión de la comunicación entre las partes implicadas pase por un intermediario común. Este centro de control se responsabiliza también del ciclo de vida de los componentes.

El “mánager” central se ha implementado en la clase **Main,** a partir del cual empieza a ejecutarse el software una vez se inicia el programa.

Lo que permite que la vista, el controlador y el modelo se puedan comunicar entre sí es una interfaz llamada **EventListener** que obliga a implementar la función “**notify**”. Mediante el uso de la clase abstracta **Event**, el **main** podrá distinguir cuál de los componentes ha notificado el evento y redireccionarlo según corresponda. Esta solución es posible ya que cada componente implementa la misma interfaz, por lo que también deben implementar su propia versión de “**notify**”, la cuál será llamada desde el **main** cuando éste decida que un evento debe ser tratado por la correspondiente parte. Por tanto, en el **main** se tendrá una instancia de cada componente de la arquitectura.

La arquitectura descrita permite una gran versatilidad a la hora de implementar los distintos elementos por separado debido a que existe un gran desacople en el sistema de comunicación. Esto permite desarrollar un módulo sin tener que pensar en cómo afectará eso en la gestión global.

VI.I MODEL

El modelo es una parte fundamental de cualquier aplicación, ya que es el componente encargado de almacenar y gestionar todos los datos que serán escritos o leídos por los usuarios, tanto de manera directa como indirecta. En otras palabras, el modelo es el lugar donde se guardan y organizan los datos que serán utilizados para realizar diferentes operaciones y análisis en la aplicación.

En el caso específico de nuestra aplicación, el modelo estaría compuesto por un array de objetos Point para representar la nube de puntos y una variable entera para representar la cantidad de puntos a almacenar.

Point

Esta clase sirve para representar los puntos que se visualizan en la nube. Una instancia de esta clase está compuesta por dos coordenadas reales para representar los valores x,y del puntos en la nube de puntos.

La clase Point reimplementa el método compareTo(Point p) de la interfaz Comparable. Este método es utilizado para llevar a cabo la ordenación de los puntos por su coordenada X en primera instancia y por su coordenada Y. Compara el punto que realiza la llamada con la instancia pasada por parámetro.

También se implementa el método distanceTo(Point p) para

calcular el valor real que representa la distancia euclidiana del punto que hace la llamada al método con respecto al punto pasado por parámetro.

ModelEvent

La clase ModelEvent hereda de **Event** y se utiliza para que el **main** notifique al Model cuando se produzca un evento que este deba gestionar. Además, aparece un enumerado ModelEventType para diferenciar el tipo de ModelEvent que se reciba. De esta manera, se simplifica la identificación de tipo de evento dentro del Modelo.

El ModelEventType compone los eventos de modelo:

- CHANGE_N_POINTS
- RESET

Los atributos de ModelEvent son el tipo para representar el ModelEvent que se haya notificado y una variable entera para almacenar la nueva cantidad de puntos que conformarán la nube de puntos.

Model

Además de la estructura array para almacenar las instancias de la clase Point y de la variable entera para guardar el número de puntos que comprenderán la nube de puntos, se encuentran dos constantes para definir los rangos de las coordenadas en los que se podrán distribuir los puntos en la nube y la variable para almacenar la referencia al **main**.

El constructor de la clase recibe el **main,** referencia al programa principal para realizar las comunicaciones entre componentes; y se inicializan las estructuras de la clase en función del entero pasado por parámetro que representa el número de puntos que conformarán la nube de puntos. A continuación, se llama a la función para inicializar la nube de puntos, initializeCloud(), la cual es la encargada de inicializar el array de puntos dándole valores aleatorios dentro del rango de las coordenadas.

Se han implementado una serie de métodos *getters* necesarios para que el controlador pueda obtener información sobre el estado del modelo.

Los métodos que resultan más interesantes para comentar son:

- getPointsRef(): devuelve la estructura que contiene la nube de puntos para que sus puntos puedan ser ordenados.

- getNearPointsRef(int mid, double d, int left, int right): devuelve una lista con los índices de los puntos que se encuentran cercanos al punto representado por el índice 'mid' dentro del rango definido por la distancia 'd', tanto para la componente x como para la y. Los índices 'left' y 'right' se utilizan para limitar los puntos a comprobar, sabiendo que fuera de estos índices no habrá ningún punto que cumpla con la condición anterior.

La clase Model implementa la interfaz **EventListener,** por lo que se debe implementar el método **notify()** donde se realizan unas operaciones u otras en función del ModelEventType recibido. CHANGE_N_POINTS, actualiza el número de puntos que aparecerán en la nube de puntos y la vuelve a inicializar; y RESET que simplemente vuelve a realizar la inicialización.

VI.II CONTROLLER

El controlador es el componente con el rol de bisagra situada entre la vista y el modelo de datos previamente explicado. Es decir, es el encargado de gestionar el flujo de datos entre los dos módulos comentados. Este flujo de datos es transformado mediante operaciones ejecutadas por el mismo controlador y visualizado en la vista o almacenado en el modelo.

En nuestra aplicación el controlador se responsabiliza de ejecutar los algoritmos correspondientes para calcular las tres parejas de puntos con menor distancia, de entre una nube de puntos 2D en un espacio equiprobable.

Antes de explicar los algoritmos, debemos tener en cuenta dos clases PointsPair y MinPairs.

PointsPair

La clase PointsPair contiene dos referencias a los puntos que componen la pareja y el valor real que representa la distancia entre ellos.

Existe un método estático para crear una instancia con distancia máxima a modo de centinela, útil para la gestión en el controlador.

En esta clase también se implementa compareTo() para ordenar de forma decreciente las instancias en función de la distancia.

MinPairs

Un objeto MinPairs está formado por una lista de instancias de PointsPair y el valor entero que representa el número de instancias almacenadas en la lista. La ejecución del algoritmo

devolverá una instancia de esta clase con las parejas de distancia mínima.

El constructor recibe el valor entero para inicializar la lista, definiendo el número de parejas que se almacenarán.

Entre los métodos de la clase se encuentran:

- fill(): completa la lista de parejas con instancias de PointsPair centinela.
- checkPoint(PointsPair p): comprueba si una pareja de puntos ha de incluirse en la lista o no. Si p ya aparece, no se incluye. Tampoco si la distancia es mayor a la máxima de la lista. Para ello solo comprobará el primer PointsPair de la lista, debido a que la lista se encuentra ordenada por valor decreciente de la distancia.
- getIndexs(): devuelve una ArrayList con los índices de los puntos de las parejas almacenadas en la lista.

ControllerEvent

La clase ControllerEvent hereda de **Event** y se utiliza para que el **main** notifique al Controller cuando se produzca un evento que este deba gestionar. Además, aparece un enumerado ControllerEventType para diferenciar el tipo de ControllerEvent que se reciba. De esta manera, se simplifica la identificación de tipo de evento dentro del Controller.

El ControllerEventType compone los eventos de controlador:

- START
- STOP

Los atributos de ControllerEvent son el tipo para representar el ControllerEvent que se haya notificado, la instancia del tipo de algoritmo que se tendrá que ejecutar y el número de parejas de puntos que se calcularán.

Controller

En el run() del Thread, se obtiene el modelo a partir de la referencia al **main** del programa. En función del tipo de algoritmo que se haya almacenado una vez recibido el evento para empezar la ejecución, se llamará a uno de los siguientes algoritmos:

- Algoritmo n^2

Se llama al método exponentialSearch() para hacer el cálculo de las parejas de puntos con menor distancia, siendo el coste del algoritmo de $O(n^2)$, ya que se trata de un doble bucle anidado en el que se calcula la distancia de cada punto con todos los otros existentes en la nube.

Por tanto, antes de entrar en el bucle, se inicializa una instancia de MinPairs pasando como parámetro del constructor el número de parejas que deberá almacenar, valor definido por el atributo de clase 'nPairs'. Una vez dentro del doble bucle, se instancia un objeto PointsPair en el que se calcula la distancia entre los puntos definidos por los índices de los bucles, 'i' y 'j'; y se comprueba si esta nueva pareja se puede incluir en la lista de MinPairs, como una de las posibles parejas de mínima distancia.

En cada iteración aparece un sleep() en el que no se realiza ninguna pérdida de tiempo, pero que es útil para que se pueda detectar la interrupción de la ejecución del algoritmo.

Este método devuelve la instancia de MinPairs que contendrá las 'nPairs' parejas de puntos con menor distancia entre sí.

- Algoritmo nlogn

La técnica anterior siendo del orden de $O(n^2)$ resulta muy costosa. Por este motivo es interesante comprobar si se puede aplicar la ya mencionada técnica de Divide&Conquer en este problema. Son 4 los requisitos que debe cumplir el problema para que se confirme que la aplicación de la técnica es válida:

1. Existencia de caso base.
2. El problema es divisible en problemas más sencillos.
3. Las soluciones de los subproblemas son independientes unas de otras.
4. Es posible la combinación de soluciones de subproblemas para obtener la solución global.

$$T(n) = \begin{cases} O(n^k), si\ a < b^k \\ O(n^k \cdot log_b n), si\ a = b^k \\ O(n^{\log_b a}), si\ a > 1 \end{cases}$$

Figura 7: Coste. Formulación recursiva de crecimiento no constante

1. Existencia de casos base

El problema tiene 3 casos base: Cuando haya 1 punto, resultando en un conjunto de distancias nulas ya que; cuando se tengan 2 puntos, resultando en una sola distancia posible; cuando se tengan 3 puntos, resultando en 3 distancias posibles.

Como lo único que hacemos en estos bloques es devolver un objecto el coste de la operación base es de O(1).

2. El problema es divisible en problemas más sencillos

Cualquier conjunto o subconjunto de puntos puede ser divido en dos mitades. Esto implica tener que encontrar el punto que marca el corte entre las dos partes, es decir, es simplemente realizar una división. Por ende, el coste de la operación del

problema es de O(1).

3. Las soluciones de los subproblemas son independientes unas de otras

La solución de un subproblema no necesita la solución de ningún otro. Es decir, encontrar la existencia de una distancia mínima o máxima en un subconjunto no supone tener que operar con su homólogo.

4. Es posible la combinación de soluciones de subproblemas para obtener la solución global

Es posible fusionar subconjuntos. Para ello se debe comprobar si existe alguna pareja con puntos de cada subproblema con una distancia entre si menor que la de alguna de las parejas de las partes.

Esto se hace calculando las distancias entre todos los puntos comprendidos en una franja F. Si llamamos dmin a la distancia de la pareja más próxima de los dos conjuntos y p al punto de corte de estos, concebimos el tamaño de F como 2*dmin con una cota inferior de p – dmin y una superior de p + dmin. Si tenemos en cuenta que la distribución es equiprobable el número de puntos dentro de F tiende a la raíz cuadrada de todos los puntos ($\sqrt{n}$), entonces la combinación de los puntos supone ($(\sqrt{n})^2$) obteniendo un coste de O(n).

Como se cumplen estos cuatro requisitos, se confirma que es válido el uso de Divide&Conquer para aplicar en el problema.

Desde el run() se llama al método logarithmicSearch(), dónde se realiza el paso previo a la aplicación de la técnica, es decir la ordenación de los puntos de la nube; y la llamada a la función recursiva closestPairs(int left, int right, int nPairs). La ordenación se realiza con la función sort() de Java cuyo coste es de O(nlogn). Una vez la nube se encuentra ordenada, se llama al método recursivo pasándole por parámetro el índice inicial y el índice final del array que representa la nube de puntos, además de pasar también el número de parejas cercanas que se deberán encontrar y dar como solución.

Dentro de la función recursiva, lo primero que se encuentra es el mismo sleep() que para el algoritmo anterior y la llamada al constructor de MinPairs, instanciando la lista el valor de 'nPairs'. La función se puede definir en tres bloques ordenados: los casos base, la división del problema y la combinación de las subsoluciones.

Casos base

Cuando solo se tenga un punto, por lo que no se puede formar ninguna pareja y por lo tanto se usa la pareja centinela para rellenar la lista de MinPairs; cuando se tengan dos puntos, se forma su pareja calculando la distancia entre ellos y se completa la lista de MinPairs con las parejas centinela para evitar problemas de instancia null en la lista; y, finalmente, cuando se tengan tres puntos, para los que se forman tres parejas de puntos calculando las distancias entre cada uno de ellos además de rellenar también la lista con parejas centinela por el mismo motivo.

División

Al tratar la nube de puntos como un array, se puede jugar con los índices para obtener el índice de la mitad del array y así poder ir dividiendo el conjunto de puntos en dos mitades, un conjunto de puntos a la izquierda de la mitad y un conjunto de puntos a la derecha de la mitad. De cada lado se obtiene una subsolución, con las que se crea una lista de parejas mediante el objeto PointsPair, la cual se ordena para conseguir tener almacenadas en un objeto MinPairs las 'nPairs' parejas de puntos cuyas distancias son menores.

Fusión

Para empezar con la fusión de subsoluciones, se realiza la llamada a la función getNearestPointsRef(int mid, double dmin, int left, int right) del modelo. A esta función se le pasa la distancia mayor de las parejas almacenadas en el objeto MinPairs mencionado en el bloque anterior, ya que si se le pasara una de las distancias menores se estarían pasando por alto posibles casos de parejas con distancias lo suficientemente menores como para ser incluidas en la combinación. El índice 'mid' sirve para identificar al punto desde el que se realizan las comprobaciones y los otros dos índices sirven para limitar el bucle que realiza dichos cálculos. La función devuelve la lista de índices que representan esos puntos que se encuentran en una distancia menor a la indicada, tanto en componente x como en y, con respecto al punto identificado por el índice 'mid'. Iterando a través de la lista de índices, se comprueba si las parejas se han de incluir en la solución o no. Finalmente, el objeto MinPairs a retornar se completa con instancias centinelas en caso de ser necesario.

Se puede conocer cuál es el coste de la implementación de la técnica. Este se trata de un algoritmo recursivo que decrementa a velocidad no constante, ya que cada conjunto se divide por la mitad, por lo que b = 2. En cada estado recursivo se realizan como máximo dos llamadas recursivas, por lo que a = 2. La operación de fusión consiste, como ya se ha comentado antes, en una operación de coste O(n) si consideramos la equiprobabilidad de los puntos, entonces es necesario tener una k tal que O(n^K) sea el máximo coste de la fusión y por ende deducimos que k = 1. Es así que siguiendo la formulación recursiva de crecimiento no constante, tal y como se muestra en la Figura 7, obtenemos que el coste de esta metodología es O($n * \log(n)$).

El resultado obtenido tras la ejecución de uno de los algoritmos se muestra por consola y se notifica a la vista. En

caso de que la ejecución no haya podido finalizar al haber sido interrumpida por el usuario, simplemente muestra por consola que algoritmo ha sido el que se ha interrumpido y se acaba la ejecución del Thread.

La clase Controller implementa la interfaz **EventListener,** por lo que se debe implementar el método **notify()** donde se realizan unas operaciones u otras en función del ControllerEventType recibido. START, obtiene qué tipo de algoritmo será ejecutado y almacena cuántas parejas de puntos deberán ser calculadas para posteriormente lanzar el Thread de la ejecución; y STOP, para detener la ejecución del Thread.

VI.III VIEW

La vista representa la interfaz de usuario de la aplicación y se encarga de presentar los datos al usuario final. La vista no interactúa directamente con el modelo ni con el controlador, sino que se comunica con ellos a través del gestor de eventos.

CloudDisplay

El constructor de esta clase recibe como parámetro una referencia al Model para facilitar la consulta del modelo y poderlo pintar de manera adecuada. También se inicializa una ArrayList de valores enteros que almacenará los índices de los puntos de las parejas cercanas que forman parte de la futura solución.

El método paint() de la clase utiliza la técnica de doble *buffering* para visualizar la nube de puntos a través de sus índices. Los puntos se pintan de negro, excepto si aparecen en la ArrayList con los índices de la solución, caso en el que se pintan de color rojo.

ViewEvent

La clase ViewEvent hereda de **Event** y se utiliza para que el **main** notifique a la View cuando se produzca un evento que esta deba gestionar. Además, aparece un enumerado ViewEventType para diferenciar el tipo de ViewEvent que se reciba. De esta manera, se simplifica la identificación de tipo de evento dentro de la View.

El ViewEventType compone los eventos de modelo:

- SHOW_RESULT

ViewEvent únicamente necesita como atributos uno para almacenar el tipo de evento de vista y otro para contener los índices de los puntos que forman parte de las parejas obtenidas como solución del algoritmo.

View

La View envía eventos tanto al Controller como al Model. Notifica al Controller cuando se pulse el botón para que se inicie la ejecución de este y cuando se pulse el botón para detener su ejecución. Al Model se le notifica cuando se modifique el número de puntos que tendrá la nube y cuando se quiera reiniciar la nube dada.

La vista es responsable de presentar los datos de manera clara y coherente para el usuario. Esto incluye la disposición de la información, la selección de la paleta de colores, la presentación de los datos en diferentes formatos y la gestión de la interacción del usuario, además de poder apreciar el resultado del algoritmo.

Podemos apreciar todos estos elementos en la interfaz:

1. JSpinner: se inicializa con el valor de número de puntps por *default* y al modificar su valor se notifica al modelo para actualizar el número de puntos y posteriormente refrescar la vista de la nube de puntos. No acepta valores menores de 6.

2. JComboBox: recibe como lista de elementos los costes de los algoritmos registrados. En la clase AlgorithmType se asocia cada algoritmo a su coste, por lo que bastará con formar la lista de costes dentro de la propia vista para realizar la correcta visualización en la ComboBox. Cuando se seleccione un algoritmo, se identificará cuál ha sido utilizando el índice correspondiente de la ComboBox.

3. JSlider: tiene definidos unos valores máximos y mínimos para elegir el número de parejas de puntos a calcular. Al modificar su valor, se muestra en el campo de texto situado encima suyo, y ya será cuando se tenga que ejecutar el algoritmo cuando se notificará el valor del Slider.

4. JButton (Start): se permite pulsar el botón mientras no haya ninguna ejecución del algoritmo iniciada. Cuando el botón sea pulsado, enviará los correspondientes eventos al modelo y al controlador, asegurando la consistencia del resto de elementos de la vista.

5. JButton (Reset): notifica al Model que debe hacer el *reset* de la nube de puntos y posteriormente manda el *refresh* del *display* de la nube. Igual que el botón de *start*, se asegura de la consistencia del resto de elementos de la vista.

6. CloudDisplay: para visualizar la nube de puntos.

Los elementos de la vista previamente descritos se pueden

observar en la Figura 8.

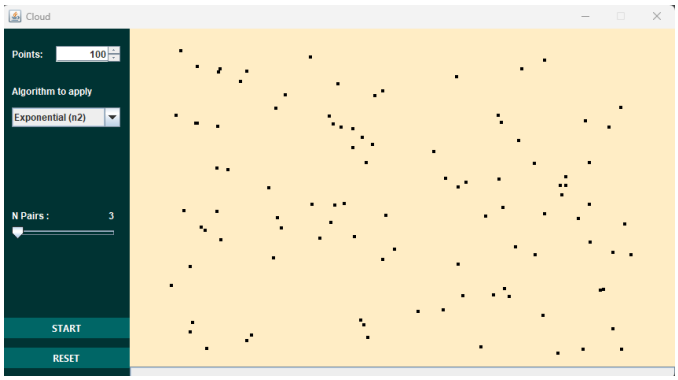

Figura 8: Vista de la aplicación

La clase View implementa la interfaz **EventListener,** por lo que se debe implementar el método **notify()** donde se realiza la operación para mostrar los resultados de la ejecución del algoritmo. Se muestra un panel con un mensaje que incluye los índices de los puntos que forman parte de la solución y se notifica a CloudDisplay para marcar dichos puntos.

VII. CONCLUSIÓN

El equipo considera que la implementación del algoritmo de *D&C* de la práctica utilizando el patrón de diseño MVC ha sido exitosa en términos de estructura y eficiencia. Pensamos que el uso del MVC junto a su consecuente abstracción ha sido bastante mejor en comparación a la realizada en las practicas anteriores. Esto nos ha permitido separar la lógica del algoritmo de la interfaz de usuario sin interferir unas con otras, lo que ha resultado en un código más organizado y fácil de mantener. Además, la utilización de *D&C* ha permitido encontrar la solución para un problema que requería una exploración exhaustiva de todas las posibilidades, de una manera menos costosa e igual de eficiente. En general, la combinación de ambos elementos ha resultado en un programa robusto y escalable.

Conseguimos acabar nuestro tercer trabajo juntos, en el que hemos podido demostrar ya nuestra comprensión de la implementación del MVC, y donde el equipo ha funcionado bastante bien. El control de versiones mediante GitHub ha sido determinante, mejorando nuestros conocimientos de la plataforma y del comando *git* sobre todo. Por los problemas que respectan a esta práctica, la comprensión del algoritmo de *D&C* fue problemático al principio, a pesar de ello, creemos que el resultado final conseguido por el equipo ha sido bastante satisfactorio.

VIII. ENLACES

1. Teoría del patrón MVC: https://developer.mozilla.org/en-US/docs/Glossary/MVC
2. Teoría del patrón por eventos: https://www.tibco.com/es/reference-center/what-is-event-driven-architecture
3. Ventajas y desventajas del patrón por eventos: https://es.wikipedia.org/wiki/Arquitectura_dirigida_por_eventos
4. Divide And Conquer: https://www.freecodecamp.org/espanol/news/significado-del-algoritmo-divide-y-venceras/
5. Coste Computacional: https://upcommons.upc.edu/bitstream/handle/2117/96
6. Merge-Sort: https://www.geeksforgeeks.org/merge-sort/991/1400228195.pdf?sequence=1&isAllowed=y
7. QuickSort: https://www.geeksforgeeks.org/quick-sort/