Tarea 3. Cálculo de la mínima distancia en una nube de puntos

Miruna Andreea Gheata, Rafael Adrián Gil Cañestro

Resumen—En este documento se muestra la implementación de la práctica *Mínima distància entre dos punts qualsevol d'un núvol de punts*. Se explicará la estructuración del proyecto, el patrón de programación empleado para el desarrollo, la solución y los distintos problemas encontrados durante el proceso y, finalmente, un manual de usuario.

Index Terms—divide & conquer, quicksort, mergesort, bucketsort, distancia euclideana

1. Introducción

En este articulo se detallará por capítulos la implementación de un programa en Java que encuentre la distancia mínima entre puntos dado una Nube de puntos.

En el capítulo 2 **se presentará el problema propuesto**, al igual que los requerimientos definidos que se tienen que cumplir.

En el capítulo 3 **se expondrá la solución propuesta**, mencionando a grandes rasgos las técnicas y patrones principales que se han adoptado para resolver el problema: el *patrón MVC*, el *DDD*, la *concurrencia*; y mencionando aspectos importantes de la implementación.

En el capítulo 4 **se especificarán las tecnologías utilizadas**, es decir, el entorno de programación, librerías destacadas, etc.

En el capítulo 5 se explicará la **implementación del Modelo**, separado en Dominio (la definición de los datos) e Infraestructura (las operaciones que se realizan sobre estos datos).

En el capítulo 6 se explicará la **implementación de la Vista**, que se ha separado en varias clases, cada una representando un elemento gráfico.

En el capítulo 7 se explicará la **implementación del Controlador**, explicando los métodos que contiene.

En el capítulo 8 se calculará el **Coste computacional del programa**, al igual que se enseñará cómo varía el tiempo de ejecución según la complejidad elegida, del número de puntos de la Nube y de los algoritmos de ordenación empleados.

El capítulo 9 corresponde a una **Guía de usuario** en la que se definen los elementos del programa y las distintas formas de interacción que puede realizar el usuario, los pasos a seguir para poder ejecutar el programa y unos ejemplos de ejecución.

En el capítulo 10 se exponen los **Problemas encontrados** a lo largo de la implementación del programa.

Y, finalmente, el capítulo 11 contiene las **Conclusiones** obtenidas tras realizar este proyecto.

2. Definición del problema

Se debe desarrollar un programa que cree una Nube de puntos y que determine cuál es la mínima distancia entre los puntos. Se debe poder realizar dos tipos de ejecuciones: una que tenga un coste computacional de $O(n^2)$ y otra que tenga un coste de $O(nlogn)$.

3. Solución propuesta

La solución planteada hace uso de distintos patrones, técnicas y formas de programación y de estructuración de proyectos.

En primer lugar, la complejidad del proceso de búsqueda vendrá determinada por lo que elija el usuario. El proceso será ***iterativo*** cuando el algoritmo elegido sea de coste $O(n^2)$, y será ***recursivo*** cuando se haya seleccionado uno de los algoritmos de costes menos elevados ($O(2 * nlogn)$ y $O(nlogn)$).

En el caso de los algoritmos recursivos se ha aplicado el método se programación conocido como ***Divide & Conquer***.

Divide & Conquer: método basado en la resolución recursiva de un problema dividiéndolo en dos o más subproblemas similares. El proceso continúa hasta que éstos llegan a ser lo suficientemente sencillos como para que se resuelvan directamente, o hasta que se llegue al caso base. Finalmente, se combinan todos los subproblemas y sus respectivas soluciones para dar una solución global.

Además, en las soluciones recursivas los puntos se ordenan según sus ejes, y para poder realizar la ordenación se dispone de **4 tipos de algoritmos de ordenación** distintos:

1. ***Javasort***
2. ***Quicksort***

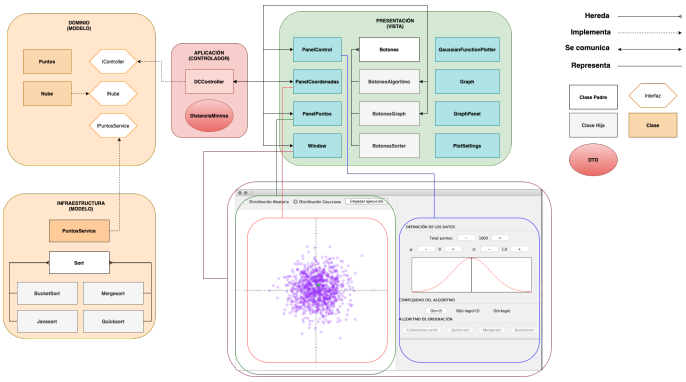


Figura 1: Estructura MVC del proyecto

3. ***Mergesort***
4. ***Bucketsort***

La implementación del programa está hecho siguiendo el conocido ***patrón MVC***.

La nube de puntos que será analizada se creará siguiendo las especificaciones del usuario. Los puntos pueden seguir **2 tipos de distribuciones**:

1. ***Distribución aleatoria***: dicha distribución hace que los puntos que se creen estén acotados por un rango inferior y superior, pero los valores que estos cojen son totalmente aleatorios.

2. ***Distribución Gaussiana***: los puntos tienen una media y una varianza definida por el usuario.

3.1. Patrón MVC y enfoque DDD

Este proyecto esta estructurado basándose en el diseño guiado por el dominio, en adelante DDD, ya que se complementa muy bien con MVC. El DDD no es una tecnología ni una metodología, es un enfoque que proporciona una visión estructural del sistema. DDD separa el *Modelo* en tres partes: ***Dominio, Aplicación e Infraestructura***.

La capa de ***Aplicación*** es responsable de coordinar la ***Infraestructura*** y la capa de ***Dominio*** para hacer una aplicación útil. Por lo general, la capa de *Aplicación* usaría la *Infraestructura* para obtener los datos, consultaría el *Dominio* para ver qué se debería hacer y luego volvería a usar la *Infraestructura* para realizar las operaciones pertinentes para obtener los resultados deseados. [1]. La capa de *Aplicación* por lo tanto se correspondería al *Controlador*, la de *Dominio* e *Infraestructura* al *Modelo*, y la de *Presentación* a la *Vista*.

3.2. Estructuración DDD

A nivel interno el programa está dividido en 4 carpetas (*Presentación*, *Aplicación*, *Dominio*, *Infraestructura*) y un fichero Main.java.

3.3. Concurrencia

El programa contiene distintos elementos de carácter concurrente con el fin de agilizar los cálculos y no bloquear la GUI. De esta manera, se tiene:

- La creación de los puntos se realiza con la implementación de un `Worker`, y los puntos que se crean se devuelven en un elemento `Future`

- La clase *PuntosService* extiende la clase `Runnable` de Java, y el proceso de cálculo se realiza creando un Thread y realizando las operaciones dentro de de `run()`.

3.4. Aspectos a destacar

1. Se ha hecho uso de clases Abstractas (***Botones*** y ***Sort***) en las que se definien atributos y métodos comunes que utilizan más de una clase. Esto se ha hecho para poder hacer un uso más inteligente de la Herencia, ahorrando así la duplicidad de código.

2. Se han extraído todas las variables utilizadas en el programa en una clase denominada ***Variables*** con el fin de centralizar todas las variables y de esta manera si se quiere modificar el valor será mucho más rápido encontrar la variable. Los nombres utilizados son fáciles de interpretar y de relacionar con los elementos que representan dentro del proyecto.

3. Se ha implementado un gráfico de densidad mediante el cual se podrá ver el comportamiento de los datos en el caso de que sigan una distribución gaussiana.

4. TECNOLOGÍAS UTILIZADAS

4.1. Lenguaje de programación utilizado

La implementación de este proyecto se ha hecho utilizando como lenguaje de programación Java.

4.2. Entorno de programación

La práctica se ha desarrollado en el IDE **IntelliJ IDEA**. **Cabe destacar que si se intenta ejecutar el proyecto el Netbeans no compilará debido a que IntelliJ utiliza carpetas extra (como la de .idea) para poder compilar el proyecto.** Por lo tanto, para ejecutar el proyecto se debe utilizar necesariamente IntelliJ.

Al igual que Netbeans, IntelliJ ofrece una ayuda a la hora de programar elementos gráficos, por lo que se ha facilitado todo el tema del diseño de las ventanas.

4.3. Librerías destacadas

`javax.swing` - se ha utilizado para poder implementar los elementos gráficos: las ventanas, los paneles, los mensajes, los botones, etc.

5. Implementación del Modelo

El *Modelo* es el elemento responsable de mantener la información con la que el sistema opera, crea nuevos puntos, reordenaremos los puntos y calcularemos las respectivas distancias. El Modelo también es el encargado de enviarle la información a la Presentación (vista), para que esta pinte por pantalla los botones, barra y gráficos. Por otro lado, las peticiones de actualización o manipulación de los datos llegan al Modelo desde el Controlador. Como ya se ha comentado las funciones del Modelo están dividas en dos partes. El Dominio se encarga mantener, enviar y modificar la información y estructura de la nube de puntos, puntos y las variables de ellas. Por el otro lado, la Infraestructura es la encargada de realizar las operaciones de ordenación y cálculo de distancia, en este caso usamos divide y vencerás, para encontrar la solución.

5.1. Dominio

Al ser el encargado de mantener y brindar la información, tiene que contener la información sobre los objetos que se van a usar y las interfaces que usamos para facilitar los futuros cambios. Dentro del Dominio, por lo tanto, podemos encontrar las definiciones de las distintas estructuras de datos que se usarán para poder representar los elementos del problema, que son Punto y Nube de puntos, tendremos también las variales que usarán las funciones del programa y una carpeta con las interficies que son IController, INube e IPuntosService.

1. La clase *Punto* contiene la definición de lo que es un punto. En nuestro caso, un punto es la composición de dos enteros que marcan su posición en los ejes x e y del gráfico, un identificador para facilitar la gestión de la gran magnitud de puntos y un booleanos isSolucion, que en el caso de ser true, ese punto pertenecerá a la pareja con la mejor distancia del conjunto total. Contiene las funciones get de todos sus atributos, un método toString para poder imprimir por pantalla las coordenadas del punto.Solo contiene un método propio
   - **`calcularDistanciaEuclidea(Punto punto)`**, que nos calcula la distancia entre el punto que se le pasa por parámetro y el punto de la clase.
2. La clase Nube que se encarga de la creación de la nube de puntos, crea la nube de puntos de forma aleatorio o gaussiana según las preferencias del usuario. Esta clase también es la encargada de implementar los métodos de creación de puntos. Nube implementa la interfaz INube. Nos encontramos que una Nube se compone de:
   - Un entero cantidad que nos indica cuantos puntos habrá en la nube
   - Un array Puntos que contiene todos los objeto Punto creados.

   Pero en la misma clase tenemos los valore máximos y mínimos que pueden tener los valores x e y.
   Los metodos que implementa de la interfaz INube son:
   - **`public Future<Nube>crearNubePuntos(int cantidad, double mediaX, double desviacionX, double mediaY, double desviacionY)`** este método crea una nube de puntos siguiendo una distribución gaussiana para ello necesita las medias de los x e y y sus respectivas desviaciones. Este método ira llamando en cada iteración a generateRandomCoordinate dos veces una para las x y otra para la y para añadir los puntos a la nube.
   - **`public Future<Nube>crearNubePuntosRandom (int lower, int upper)`** crea puntos de forma aleatoria solo debe tener dos limites para saber sobre que rango deben estar los puntos.

   Por otro lado, tiene estos metodos que son propios de la clase y no se encuentran en la interfaz:
   - **`private double generateRandomCoordinate(double media, double desviacion)`** es método es el generador de coordenadas gaussianas con la media y la desviación nos creara un double aleatorio.
   - Contiene además, por un lado, los gets de el array de puntos, la cantidad de puntos, y el rango máximo y mínimo de los ejes x e y. Por el otro lado, los sets de el array de puntos y un punto.
3. La clase *Variables* contiene todas las "variables"que se usan en el programa. Aquí podemos encontrar las medidas de la ventana, los valores de los botones, los textos que apareceran en la ventana y los colores de los puntos y los puntos solución.
4. Por último, encontramos la carpeta con las tres interfaces que son: IController, INube e IPuntosService que son usados por las clases DCController, Nube y PuntosService respectivamente. Las funciones incluidas en cada interfaz se comentan en las secciones de las clases anteriores.

5.2. Infraestructura

La infraestructura es la encargada de hacer todos los cálculos de las distancias entre los puntos y dar la pareja solución del problema. El problema puede variar de distintas formas una de ellas es si usamos un algoritmo de divide y vence. De estos usamos dos cada uno con un coste asociado distinto. Si no usamos ningún el algoritmo el coste de $O(n^2)$, pero si usamos uso de los dos algoritmos el coste va de$O(nlogn)$ a $O(2nlogn)$. La diferencia principal es que el primero es iterativo y los otros dos son métodos recursivos.

En la infraestructura nos encontramos dos archivos java:

1. *SortingTypes* que contiene un enumerado de los 4 tipos de ordenación que utilizamos:
   - Javasort
   - Mergesort
   - Quicksort

- Bucketsort

2. *PuntosService* es la clase encargada de realizar la búsqueda de la distancia mínima entre los puntos de la nube. El proceso de cálculo se realiza mediante un Thread para agilizar los cálculos y así no bloquear a la GUI. Podemos observar que el contructor de esta clase solo contiene el controlador. Además, *PuntosService* implementa la interfaz *IPuntosService*, que sus métodos serán comentados más adelante.

En primer lugar, las variables que contiene esta clase son:

- sorter tipo Sort, usado para guardar el tipo de mezclador que ha elegido el usuario.
- worker tipo Thread, es el hilo encargado de llamar a todas las funciones.
- nubePuntos tipo Nube, es la nube de puntos con la que trabajan los algoritmos.
- algoritmoElegido tipo int, guarda que tipo de coste ha elegido el usuario.
- isPuntosOrdenados tipo boolean, en caso de ser true, nos indica que el Array de puntos está ordenado.
- arrayPuntosOrdenados tipo Array List, usado por los algoritmos recursivos para tratar los puntos
- puntosOrdenadosCoordenadaX es un Array de Punto que se usa como parametro en las llamadas recursivas
- puntosOrdenadosCoordenadaY es un Array de Punto que se usa como parametro en las llamadas recursivas
- controller tipo DCController, encargado de la comunicación desde la infraestructura

Los métodos que utiliza la clase son:

`void setClaseSort(String p)`: inicializa el tipo de mezclado, dependiendo de la elección del usuario.

`void setNubePuntos(Nube nubePuntos)`: inicializa la nube de puntos con la que se van a calcular las distancias.

`void start()`: lanza el hilo encargado de hacer los calculos dependiendo de las características que haya elegido el usuario.

`run()`: cuando se lanza *start*, *run* se ejecuta. Dentro tiene un switch que dependiendo de *algoritmoElegido*, se ejecutara uno de los 3 casos posibles `case 0: naive`, `case 1: divideConquerOnlogn2` o `case 2: DistanciaMinima divideConquerOnlogn`.

`DistanciaMinima naive(Punto[] puntos, int n)`: Método iterativo que calcula la distancia entre los puntos. Es de coste (n^2) ya que mira todos los puntos con todos los demás.

```
DistanciaMinima distanciaMinima = null;
double min = Double.MAX_VALUE;
double computedDistance;
for (int i = 0; i < n; i++){
    Punto puntoActual = puntos[i];
    for (int j = i + 1; j < n; j++){
        computedDistance = puntoActual.calcularDistanciaEuclidea(puntos[j]);
        if (computedDistance < min){
            min = computedDistance;
            distanciaMinima = new DistanciaMinima(puntoActual, puntos[j], min);
        }
    }
}
return distanciaMinima;
```

`DistanciaMinima divideConquerOnlogn2(Punto[] puntos, int n)`: Método recursivo que busca la distancia mínima aplicando el concepto de Divide Conquer. Divide de manera recursiva el total de puntos y busca para cada mitad la solución local, al salir de la recursividad se encontrará la solcuión global.

```
if (n <= 3){
    return naive(puntos, n);
}

int mid = n / 2;

Punto puntoMedio = puntos[mid];

Punto[] puntosXIzquierda = Arrays.copyOfRange(puntos, 0, (n + 1)/2);
Punto[] puntosXDerecha = Arrays.copyOfRange(puntos, n/2, n);

DistanciaMinima distanciaMinimaL = divideConquerOnlogn2(puntosXIzquierda, mid);
DistanciaMinima distanciaMinimaR = divideConquerOnlogn2(puntosXDerecha, n - mid);

DistanciaMinima distanciaMinima = min(distanciaMinimaL, distanciaMinimaR);

ArrayList<Punto> stripList = new ArrayList<>();

int total = 0;

for (int i = 0; i < n; i++){
    double distPuntos = distanciaMinima.getDistanciaPuntos();
    if (Math.abs(puntos[i].getX() - puntoMedio.getX()) < distPuntos)){
        stripList.add(puntos[i]);
        total++;
    }
}
double distPuntos = distanciaMinima.getDistanciaPuntos();
DistanciaMinima distMedio = stripClosest(stripList, total, distPuntos);
return min(distanciaMinima, distMedio);
```

`DistanciaMinima divideConquerOnlogn(Punto[] puntosX, Punto[] puntosY, int n)`: Método recursivo que busca la distancia mínima aplicando el concepto de Divide Conquer. Divide de manera recursiva los puntos y calcula las soluciones locales para después encontrar la solución global. A diferencia del anterior este tiene los puntos ordenador tanto por eje X como por eje Y.

```
if (n <= 3){
    return naive(puntosX, n);
}

int mitad = n / 2;

Punto puntoMedio = puntosX[mitad];

ArrayList<Punto> puntosIzq = new ArrayList<>();
ArrayList<Punto> puntosDer = new ArrayList<>();

for (int i = 0; i < puntosY.length; i++){
    if (puntosY[i].getX() <= puntoMedio.getX()){
        puntosIzq.add(puntosY[i]);
    } else {
        puntosDer.add(puntosY[i]);
    }
}

Punto [] puntosYIzquierda = toArray(puntosIzq);
Punto [] puntosYDerecha = toArray(puntosDer);

Punto[] puntosXIzquierda = Arrays.copyOfRange(puntosX, 0, (n + 1)/2);
Punto[] puntosXDerecha = Arrays.copyOfRange(puntosX, n/2, n);

DistanciaMinima distL = divideConquerOnlogn(puntosXIzquierda, puntosYIzquierda, mitad);
DistanciaMinima distR = divideConquerOnlogn(puntosXDerecha, puntosYDerecha, n - mitad);

DistanciaMinima distanciaMinima = min(distanciaMinimaL, distanciaMinimaR);

ArrayList<Punto> stripList = new ArrayList<>();

int puntosStrip = 0;

double min = distanciaMinima.getDistanciaPuntos();

for (Punto punto : puntosY) {
    if (Math.abs(punto.getX() - puntoMedio.getX()) < min) {
        stripList.add(punto);
        puntosStrip++;
    }
}

DistanciaMinima distMedio = stripClosest(stripList, puntosStrip, min);
return min(distanciaMinima, distMedio);
```

`DistanciaMinima stripClosest(Punto[] strip, int cantidad, double distanciaPuntos, boolean isNLogN)`: Método que se encarga de encontar una distancia menor a la ya encontrada en las dos mitades.

Esta distancia se puede encontrar entre los puntos que están a una distancia menor a la encontrada en el array.

ordenarPuntosPorCoordenadas(Nube puntos): ordena los puntos de la nube que entra por parámetros dependiendo de sus coordenadas llamando al método sort.

Punto[] toArray(ArrayList<Punto>pts): transforma la lista que entra por parámetro en un Array de puntos

DistanciaMinima min(DistanciaMinima d1, DistanciaMinima d2): de las dos distancias que le entran por parámetro devuelve la de menor valor.

setAlgoritmoElegido(int algoritmoElegido): establece el algoritmo para encontrar la distancia mínima. La entrada puede tener el valor 0, 1 o 2.

Por último tenemos la carpeta SortingAlgorithms, que contiene todos los algoritmos utilizados en la que encontramos las clases:

1. La clase Sort que es la clase padre de las 4 que utilizamos en el programa. La clase contiene las variables:
   - puntos Array de *Punto* que es con los que trabajarán los algorimos.
   - nombreAlgoritmo tipo String, usamos para identificar al algoritmo
   - duración total tipo long, usado para guardar el tiempo que tarda el algoritmo en teminar la ejecución
   - mirarCoordenadaX tipo boolean para que los algoritmos sepan que coordenada deben ordenar
   - puntosOriginales Array de *Punto* con tiene los puntos sin ningún tipo de ordenación

   Y los métodos:
   - **void sort(Punto[] puntos)** en este método irá implementado el algoritmo de ordenación de cada clase hija
   - **setPuntosOriginales(Punto[] puntosOriginales)** inicializa los puntos con los que se trabajara
   - **setMirarCoordenadaX(boolean mirarCoordenadaX)**
2. La clase Buckersort implementa esté sort:

```
    this.puntos = puntos;
List<Integer> idOrdenados = sort(Arrays.asList(puntos));

int index = 0;

for (Integer id : idOrdenados) {
    puntos[index] = puntosOriginales[id - 1];
    index++;
}
```

   Este método se complementa con las funciones:

List<Integer>concatBuckets(List<List<Integer buckets): une todos los buckets ordenados y retorna la lista de puntos ordenados

List<List<Integer splitIntoUnsortedBuckets(List<Integer>initialList): se encarga de dividir el conjunto de puntos en buckets con el fin de facilitar la ordenación del conjunto. Los puntos se dividen mediante una función de hash utilizando el id de los puntos.

static int hash(double i, double max, int numberOfBuckets): devuelve el número del "cubo" o posición a la que pertenece ese elemento.

List<Integer>sort(List<Punto>arrayToSort): llanza el proceso de division del array con *splitIntoUnsortedBuckets* y después los vuelve a unir llamando a *concatenateSortedBuckets*.

La clase Javasort implementa esté sort:

```
    this.puntos = puntos;
duracionTotal = System.nanoTime();
sortPrivate();
duracionTotal = System.nanoTime() - duracionTotal;
```

Este método se complementa con las funciones:

void sortPrivate(): compará todos los puntos dependiendo de la coordenada que se elija.

La clase Mergesort implementa esté sort:

```
if (l < r)
{
    // Find the middle point
    int m = (l+r)/2;

    // Sort first and second halves
    sort(arr, l, m);
    sort(arr , m+1, r);

    // Merge the sorted halves
    merge(arr, l, m, r);
}
```

Este método se complementa con las funciones:

void merge(Punto arr[], int l, int m, int r): que vuelve a unir en un mismo array los elementos que se habian dividido en las llamadas recursivas anteriores.

La clase Quicksort implementa esté algoritmo:

```
 duracionTotal = System.nanoTime();
high = puntos.length - 1;
low = 0;
this.puntos = puntos;
sort(low, high);
duracionTotal = System.nanoTime() - duracionTotal;
```

Este método se complementa con las funciones:

void sort(int low, int high): mediante un pivote va reubicando los elementos del array dependiendo de si el número es mayor que el pivote o menor, los elementos se van cogiendo desde los extremos hacia el elemento central del array cuando se crucen el algoritmo acaba.

6. Implementación de la Vista

La *Vista* es la responsable de representar gráficamente el modelo y, además, facilitar la interacción del usuario con el programa para que pueda:

- definir los datos del modelo, en este caso los puntos.
- conocer el estado del programa (mensajes de error, el estado de ejecución del proceso...)
- conocer la solución del problema

Con el fin de llevar esto a cabo, se ha dividido la Vista en distinta clases. Siguiendo una organización jerárquica, se tienen:

1. ***Window***: Ventana principal que sirve como contenedor de todos los elementos gráficos. Cabe destacat el elemento `JOptionPane` que se usa para poder mostrar mensajes al usuario. Hay 2 tipos de mensajes que se pueden mostrar: el mensaje de error (que se puede dar en 2 casos, como se puede apreciar en [2])

2. ***PanelControl***: Panel que contiene todos los elementos de control del programa. Aquí se pueden encontrar los elementos
   - ***BotonesAlgoritmo***, que contiene los botones con los que se puede seleccionar el tipo de coste asintótico que se desea tener
   - ***BotonesGraph***, que contiene los controles de la distribución gaussiana. Se pueden modificar la varianza y la media de los puntos y, además, se puede visualizar el gráfico de densidad de la distribución a medida que se van actualizando los parámetros [3]
   - ***BotonesSorter***, que contiene los botones con los que se puede seleccionar el algoritmo de ordenación que se desea aplicar. Se puede elegir el *Mergesort*, *Quicksort*, *Bucketsort* y "*Javasorsort*"(`Collections.sort()`).
   - ***Barra de progreso***. Dicha barra se actualiza de dos maneras dependiendo del tipo de algoritmo que se ha seleccionado:
     - Si se ha seleccionado el método iterativo $O(n^2)$, la barra se irá actualizando según se realicen los cálculos, de manera que cuando el proceso de búsqueda acabe la barra se habrá completado
     - Si por lo contrario se ha seleccionado uno de los métodos recursivos $O(2 * nlogn)$ o $O(nlogn)$, la barra simplemente mostrará que el proceso se está ejecutando, pero no mostrará el total de cálculos realizados, ya que es complicado determinar el número de cálculos totales que se deben hacer para llegar a la solución.

3. ***PanelPuntos***: Panel que contiene los botones para seleccionar el tipo de distribución (*Gaussiana* u *Aleatoria*), y el *PanelCoordenadas*.

4. ***PanelCoordenadas***: Panel en el que se pintan los puntos que se quieren analizar. Este panel es distinto según el tipo de distribución que se ha escogido:
   - Si se sigue una *distribución aleatoria*, el eje de coordenadas tendrá su origen en la parte inferior izquierda del panel. Se representarán sólo los ejes positivos. El rango de los ejes va de (0, 0) a (5, 5) [4a]
     .
   - Si se sigue una *distribución Gaussiana*, el eje de coordenadas tendrá su origen en el centro del panel (280, 280). Se representarán tanto los ejes negativos como los positivos. El rango de los ejes va de (-5, -5) a (5, 5) [4b].

5. ***Graph, GraphPanel, GaussianFunctionPlotter y PlotSettings***: clases que sirven para poder representar el grafo de densidad de los puntos cuando estos sigan una distribución gaussiana [3]. El código está basado en el que se puede encontar en esta página [3]

7. Implementación del Controlador

El *Controlador* es el elemento que monitoriza y controla las comunicaciones entre la Vista y el Modelo.

El controlador implementa la interfaz `IController` que tiene los siguientes métodos:

`void crearNubePuntos()`: se llama desde el controlador cuando o bien se debe crear la nube de puntos, o bien se debe modificar debido a que haya cambiado sus características. Según la distribución que los puntos sigan, llamará al método que crea los puntos de manera aleatoria o de manera controlada con media y con desviación. Este proceso es concurrente y la llamada retorna un elemento del tipo `Future<Nube>`. Una vez creados los puntos se actualizan los valores pertinentes que hacen referencia a estos.

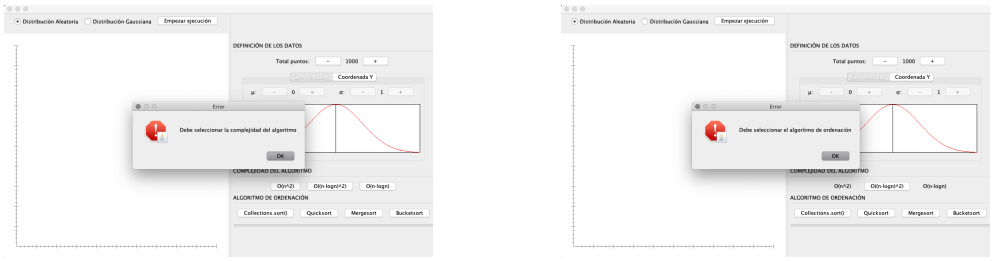
`void deshabilitarBotonesSorter()`: se llama desde *PanelControl* cuando se modifica la complejidad del proceso a $O(n^2)$. Los botones Sorter son aquellos que te permiten elegir el algoritmo de ordenación que se quiere emplear cuando se ordenen los puntos según sus coordenadas. Sólo se utilizan los algoritmos de ordenación en el caso de los algoritmos recursivos.

`void habilitarBotonesSorter()`: se llama desde *PanelControl* cuando se modifica la complejidad del proceso a $O(2 * nlogn)$ o $O(nlogn)$.

`void disableGaussianElements()`: se llama desde *PanelPuntos* elige que los puntos sigan una *distribución aleatoria*. Los elementos gaussianos se corresponded con los botones para modificar la media y la desviación de los puntos.

`void enableGaussianElements()`: se llama desde *PanelPuntos* elige que los puntos sigan una *distribución gaussiana*.

`void inicializarPuntos()`: se llama desde el *PanelPuntos* cuando se ha pulsado el botón de `Start`. Primero, el controlador comprueba si la nube de puntos no ha sido ya inicializada o si la nube de puntos se debe cambiar debido a que se hayan modificado los parámetros de los puntos (total puntos, media y desviación). Si no es así, se llama a la clase *Nube* que es la que se encarga de



(a) Mensaje de error cuando no se ha especificado la complejidad deseada

(b) Mensaje de error cuando no se ha especificado el algoritmo de ordenación

Figura 2: Mensajes de error que puede dar el programa

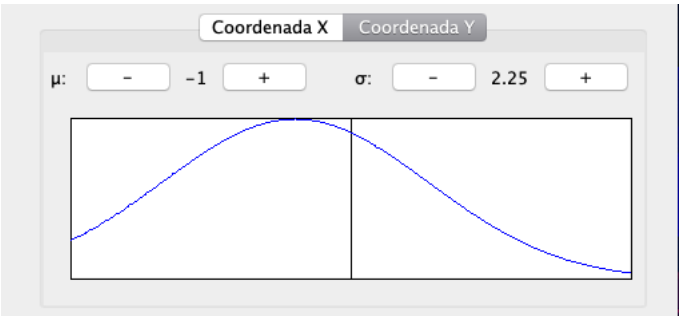


Figura 3: Elementos de control de la distribución Gaussiana

crear los puntos.

`void pintarPuntos()`: método que se llama desde el *PanelPuntos* para pintar los puntos en el *PanelCoordenadas*. Para poder representar los puntos de forma correcta se deben determinar los límites de los ejes de coordenadas. Por consecuente, lo primero que este método hace es determinar los límites de los ejes, y a continuación se llama al método `repaint()` del *PanelPuntos* para que se pinten los puntos.

`void setearParametrosElegidos()`: se llama desde el *PanelPuntos* cuando se ha pulsado el botón de `Start` para poder seteat el algoritmo de ordenación y la complejidad de cálculo que el usuario haya elegido. En caso de que no se haya elegido una complejidad salta un error y, si se ha elegido una complejidad distinta de $O(n^2)$ pero sin elegir algoritmo de ordenación, salta otro error.

`void setPuntoSolucion(DistanciaMinima distanciaMinima)`: se llama desde *PuntosService* cuando se encuentra la solución para que se pinte en el *PanelCoordenadas* y se muestre el mensaje al usuario.

`void start()`: se llama desde *PanelPuntos* cuando se pulsa el botón `Start`. Se encarga de llamar a la función `start()` del *PuntosService* para que comience el proceso de cálculo de las distancias.

`void updateGraph(double mean, double stdDeviation)`: se llama desde *PanelControl* cuando se modifica la desviación o la media de los puntos. Este método a su vez llama a *GraphPanel* para que se modifique el gráfico de la función de densidad de los puntos.

8. Coste computacional

El coste computacional del programa varía según los algoritmos de ordenación y la complejidad de cálculo que se eligen.

	Javasort	**Quicksort**	**Mergesort**	**Bucketsort**
$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$
$O(n2logn)$	$O(n2logn)$	$O(n2logn)$ - $O(n^2)$	$O(n2logn)$	$O(n2logn)$ - $O(n^2)$
$O(nlogn)$	$O(nlogn)$	$O(nlogn)$ - $O(n^2)$	$O(nlogn)$	$O(nlogn)$ - $O(n^2)$

8.1. Incremento del tiempo de ejecución según las opciones elegidas y el número de puntos

Como se puede apreciar en la figura [5] el incremento más elevado es el de la complejidad $O(n^2)$, que para 100.000 puntos tarda 22 segundos.

Para las complejidades restantes, el máximo tiempo que se tarda es de 0.125 segundos, por lo que son 99.43

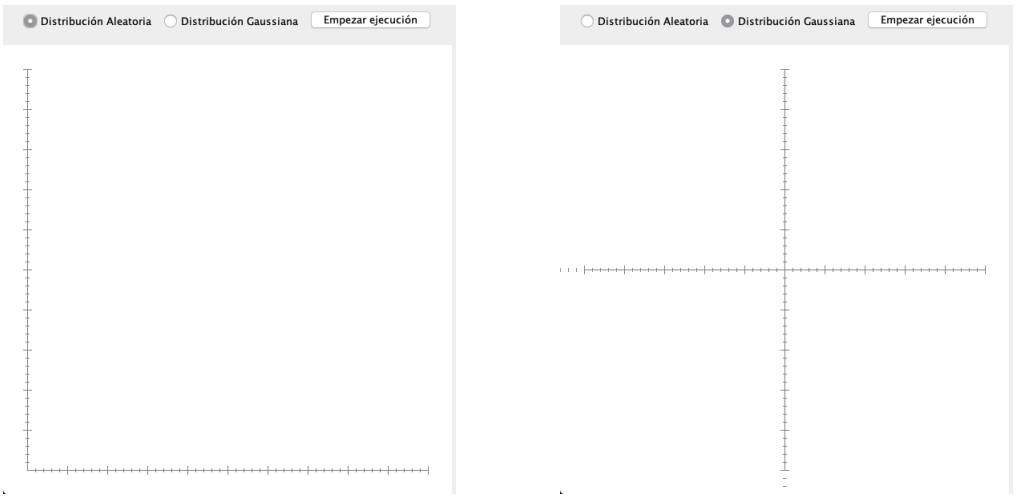
9. Guía de usuario

9.1. Definición de los componentes

Se tiene total libertad para poder manipular tanto los datos como el tipo de ejecución que se desea realizar.

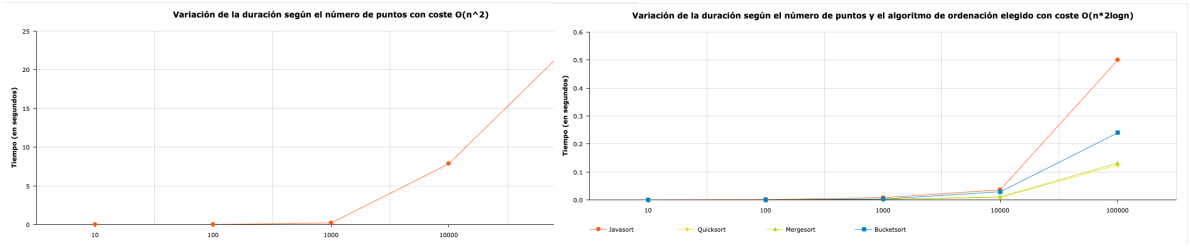
Datos: es la nube de puntos. Se puede:

- Modificar el tamaño. El rango de puntos que se puede generar va de 10 a 100.000
- Modificar la distribución. Los puntos pueden seguir una:
  1. Distribución aleatoria: los puntos se crean de forma aleatoria dentro un rango inferior y superior
  2. Distribución gaussiana: los puntos tienen una media y una desviación que se puede elegir
- Modificar las características de los puntos. Si se selecciona la distribución gaussiana se pueden modificar las medias y las desviaciones de ambas cordenadas.

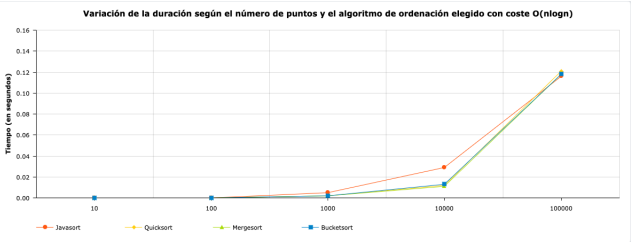


(a) Eje de coordenadas de la distribución aleatoria (b) Eje de coordenadas de la distribución gaussiana

Figura 4: Duración de la ejecución por pieza a medida que se incrementa la dimensión del tablero



(a) Complejidad $O(n^2)$ (b) Complejidad $O(n2logn)$



(c) Complejidad $O(nlogn)$

Figura 5: Duración de la ejecución por tipo de complejidad según el total de puntos de la nube

Complejidad del algoritmo: se puede elegir entre:

- $O(n^2)$: se trata de un "doble for" que calcula todas las distancias entre todos los puntos.
- $O(2nlogn)$: hace uso del principio de Divide y Vencerás. Se divide la nube mediante un proceso recursivo y se calcula la minima distancia de todas las subdivisiones. Ordena dos veces los puntos, la primera antes de dividir y la segunda cuando busca los puntos de la mitad de la nube. [1]
- $O(nlogn)$: hace uso del principio de Divide y Vencerás. Se divide la nube mediante un proceso recursivo y se calcula la minima distancia de todas las subdivisiones. A diferencia del anterior ordena los puntos sólo antes de dividir la nube. [2]

Algoritmo de ordenación: se puede elegir entre:

- Javasort: coste$O(nlogn)$
- Quicksort: coste en el mejor caso $O(nlogn)$, y en el peor $O(n^2)$
- Mergesort: coste $O(nlogn)$
- Bucketsort:coste $O(n+k)$, donde k es el número de buckets

9.2. Ejecución del programa

Para poder ejecutar el programa se debe:

1. Abrir el proyecto con IntelliJ IDEA
2. Compilar el proyecto
3. Ejecutar el programa

Una vez hecho esto aparecerá la ventana principal del programa. Para poder realizar una correcta ejecución:

1. Se debe elegir la complejidad del algoritmo en el panel derecho. Si se elige una complejidad distinta de $O(n^2)$ se debe elegir también el algoritmo de ordenación, ya que si no se hace la ejecución no podrá seguir (como se puede observar en [6b])

2. Si se desea modificar la distribución de los puntos se debe elegir una de las opciones del panel de la derecha: ***Distribución Aleatoria*** o ***Distribución Gaussiana***. Por defecto está seleccionada la distribución aleatoria.

   En el caso que se seleccione la distribución gaussiana, se habilitarán los botones que permiten modificar la media y la varianza de las coordenadas [3].

3. Para ejecutar el programa se debe pulsar el botón `Empezar ejecución` de la parte superior del panel.

   Durante la ejecución, en los casos que dura más tiempo, la barra de progreso se actualizará de dos formas dependiendo de si es un proceso iterativo, recursivo, o muy costoso:

   - En el proceso iterativo (complejidad $O(n^2)$) la barra de progreso se actualiza en % hasta que se completa. **Excepción:** en el caso del proceso iterativo con 100.000 puntos se ha puesto la misma barra que en la recursiva

   - En el proces recursivo la barra de progreso va mostrando que "está calculando", pero no se conoce el tiempo restante que va a durar. En los casos de los algoritmos de complejidad $O(nlogn)$ y $O(2nlogn)$ no se aprecia como se carga ya que no tienen una duración muy elevada, pero en el caso de $O(n^2)$ sí.

4. Finalmente, la solución aparecerá con una ventana de mensaje en la que se especifarán los puntos con sus coordenadas, la duración total de la ejecución y la distancia entre dichos puntos. Además, en el grafo de puntos la solución aparecerá con verde.

5. Para volver a ejecutar el programa se debe cerrar la ventana de la solución y seleccionar otros parámetros si se desea y volver a darle al botón de `Empezar ejecución`.

9.3. Ejemplos de ejecuciones

En la figura [6] se pueden observar dos tipos de ejecuciones: una que lleva a una solución [6a] y otra que lleva a los mensajes de error [6b].

10. Problemas encontrados

10.1. Uso de JavaFX

En un principio, para poder representar los puntos se ha considerado la librería de `javafx` ya que visualmente ofrecía un aspecto bastante atractivo y, además, sus clases `ScatterChart` simplificaba mucho la representación de los puntos en los ejes. Sin embargo, con un número elevado de puntos (100.000) la GUI se quedaba bloqueada y, además, afectaba el tiempo de la ejecución total del programa, por lo que en un final se ha optado por implementar el grafo de los puntos sin el uso de ninguna librería y mediante componentes de `swing`.

10.2. Implementación del Bucketsort

El Bucketsort es conocido por colocar mediante una función de hash los elementos en los buckets para después ordenarlos. Teniendo como elementos los elementos de tipo *Punto*, se ha tenido que mirar cuál tendrá que ser el criterio de colocación en los buckets y el criterio de comparación de los elementos.

Se ha optado por añadir un atributo id a la clase *Punto* y usar este como elemento para poder distribuir los puntos en los buckets, y para poder ordenar los elementos se tendrían en cuenta una de las dos coordenadas de los puntos.

10.3. Implementación de la barra de progreso

Para los métodos iterativos la barra de progreso ha sido sencillo de implementar ya que se conocía que el número máximo de pasos que se harán son $(n*(n-1))/2$.

La barra de progreso para los métodos recursivos no se ha podido implementar de la forma de "% trabajo realizado" porque no es posible conocer el número de llamadas recursivas que se harán en total durante la ejecución, ni tampoco se nos ha ocurrido la forma lógica de contar los pasos totales de la ejecución. Por lo tanto, se ha optado por tener una barra que muestre que el programa se está ejecutando pero sin saber cuánto tiempo falta para su finalización.

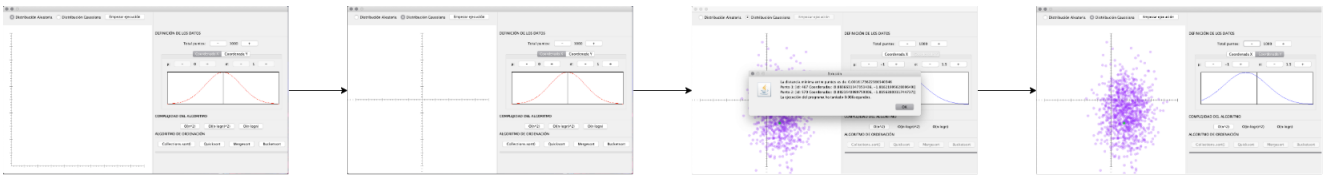
11. Conclusiones

Se ha podido observar cómo la complejidad de los algoritmos afecta en el tiempo de ejecución del programa.

Por lo tanto, es algo a lo que le debemos prestar mucha atención y estudiar de forma exhaustiva cual debemos usar. Además, poder trabajar con paquetes de datos de tamaños tan diversos como los que hemos usado en esta práctica, nos da una perspectiva de como pueden ser los problemas que nos podamos encontrar en nuestro futuro laboral.

Referencias

[1] *Closest Pair of Points using Divide and Conquer algorithm* https://www.geeksforgeeks.org/closest-pair-of-points-using-divide-and-conquer-algorithm/
[2] *Closest Pair of Points | O(nlogn) Implementation* https://www.geeksforgeeks.org/closest-pair-of-points-onlogn-implementation/
[3] *Código para gráfico de densidad* http://vase.essex.ac.uk/software/JavaPlot/code/ac/essex/graphing/charts/



(a) Ejecución con 1000 puntos de distribución gaussiana

(b) Ejecución incorrecta del programa que lleva a los mensajes de error

Figura 6: Ejecución con 1000 puntos de distribución gaussiana y con un coste de $O(n^2)$