Práctica Capítulo 3.- Pareja de puntos más cercanos (2023)

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

***Resumen*—En este documento se explica nuestra implementación de la práctica "Pareja de puntos más cercanos". En este, explicamos como hemos implementado el patrón MVC, el funcionamiento de la interfaz gráfica de nuestro programa, el funcionamiento de nuestros algoritmos de Divide y Vencerás, 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*—** Divide y Vencerás, Fuerza bruta, Espacio equi-probable, Patrón MVC, Patrón por eventos.

I. INTRODUCCIÓN

En este tercer taller se ha desarrollado con patrón MVC una aplicación que calcula la mínima distancia entre dos puntos de una nube de puntos 2D. El programa es capaz de calcular las tres parejas de puntos con menor distancia, entre una nube de puntos bidimensional en un espacio equi-probable. Se comparan dos algoritmos diferentes que resuelven el problema para diferentes medidas de N. Un algoritmo sigue el esquema de fuerza bruta con un coste asintótico de complejidad $O(n^2)$ i el otro con el esquema Divide y Vencerás presenta un coste $O(n \cdot \log n)$. Además de una nube de puntos 2D con distribución uniforme, también se ha hecho lo mismo para la distribución gaussiana y chi-cuadrado. También se ha implementado el cálculo de la pareja de puntos más alejados.

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 17.

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.

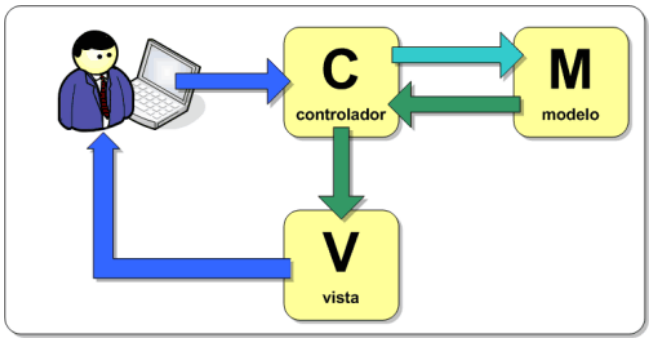

Fig. 1. Patrón MVC

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 “CALCULAR”, 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 las tres parejas de puntos más cercanos o la pareja de puntos más alejados a elección del usuario.

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. El problema del par de puntos más cercano

Es un problema común en ciencias de la computación y matemáticas que implica encontrar el par de puntos más cercano en un conjunto de puntos dados. En otras palabras, se busca encontrar dos puntos en un conjunto de puntos que estén más cerca entre sí que cualquier otro par de puntos del conjunto.

El problema del par de puntos más cercano tiene varias aplicaciones prácticas, como en la visión por computadora, el procesamiento de imágenes, la robótica y el diseño de circuitos integrados. Además, también tiene aplicaciones en teoría de grafos, optimización, geometría computacional y otras áreas de la matemática y la ciencia de la computación.

Hay varios algoritmos para resolver el problema del par de puntos más cercano, como el algoritmo de fuerza bruta y el algoritmo de divide y vencerás. Cada algoritmo tiene sus propias ventajas y desventajas, y la elección del algoritmo depende del tamaño del conjunto de puntos y la precisión requerida en la solución.

Adicionalmente en esta práctica se pide el conjunto de los tres pares de puntos más cercanos.

VI. Algoritmo divide y vencerás

Como hemos visto en los algoritmos de ordenamiento, es posible encontrar algoritmos con un coste $O(n \cdot \log n)$ para resolver un problema utilizando la técnica conocida como divide y vencerás, por lo que se plantea la manera de aplicar esta técnica a la solución de nuestro problema.

El siguiente algoritmo se basa en una implementación de esta técnica, y consiste en ordenar los puntos según una de sus coordenadas, por ejemplo x, y posteriormente utilizar esta ordenación para dividir el plano en dos mitades y así recursivamente, una vez divididos los puntos, se buscan los tres pares de puntos más cercanos en cada mitad. Definiendo las fases que forman un algoritmo tipo divide y vencerás, nuestro algoritmo debe realizar lo siguiente:

Dividir.- Ya que se supone que los puntos se almacenan en un vector P. Si P contienen pocos puntos, podemos simplemente aplicar la solución iterativa, ya que con una cantidad pequeña de puntos, podemos suponer que tarda un tiempo constante. Si nuestro array de puntos es mayor a esta cantidad, entonces trazamos una línea l que divida el plano en dos mitades con aproximadamente la misma cantidad de puntos.

Vencer.- Aplicamos el algoritmo en forma recursiva en cada mitad de puntos, obteniendo la tríada de puntos de menor distancia de la mitad izquierda del plano y de la mitad derecha, y construimos una nueva tríada seleccionando de los 6 puntos aquellos 3 con menor distancia.

Combinar.- Podría darse el caso que la tríada de puntos más cercanos tuviera puntos a ambos lados de la línea divisoria, por lo tanto debemos verificar si no hay dos puntos, uno a cada lado de la l, cuya distancia sea menor a la distancia máxima de los 6 pares de puntos (dist). Podemos ver que no es necesario comparar todos los puntos a cada lado de l, ya que si un punto se encuentra a una distancia mayor que dist de l, podemos asegurar que no habrá un vecino con distancia menor que dist al otro lado de l. Por lo tanto solo comparamos los puntos que se encuentran a una distancia menor a dist de l.

VII. Implementación del controlador

En el controlador hemos implementado un switch que ejecuta el método que hayamos seleccionado en la interfaz gráfica. Todos los métodos quedan implementados en esta clase y explicados en posteriores apartados.

VII. Implementación del patrón MVC

Para esta práctica, hemos optado por NO realizar una implementación de MVC por eventos, sino por una implementación basada en Punteros que inicializamos en el programa principal y distribuimos entre los componentes para que se comuniquen entre ellos.

Al iniciar el Programa, se creará la Vista con todos los componentes (vista principal y sus 2 paneles laterales), sin inicializar tanto el Modelo como el Controlador.

VIII. Implementación del Modelo

Para la implementación del modelo de datos hemos usado la clase **Model.java**. Contendrá todos los datos usados para la resolución del Algoritmo de Dividir y Vencer y almacenará la solución una vez el controlador haya acabado su ejecución.

En esta clase, contendremos todos los datos necesarios para la resolución del algoritmo de Dividir y Vencer y guardará la solución.

Los atributos de la clase son:

- **vista:** puntero a la vista.
- **controlador:** puntero al controlador.
- **N:** número de puntos en el plano 2D.
- **puntos:** array de puntos generados.
- **soluciones:** parejas que forman la solución.
- **distancias:** array que almacena la distancia entre los puntos de la solución.
- **distribución:** enumerado con la distribución para generar los puntos.
- **metodo:** enumerado con los diferentes algoritmos para resolver el problema.
- **minimizar:** booleano para gestionar la opción de buscar la distancia mínima o la distancia máxima entre puntos del plano 2D.
- **cantidadParejas:** número de parejas guardadas con sus distancias.
- **ANCHO, ALTO:** dimensiones de la ventana de representación.

Además de los atributos, tendremos dos constructores y varios métodos que usarán otras clases (o la misma clase). Estos métodos son:

métodos privados:

- **generarDatos():** método que genera aleatoriamente los puntos según la distribución elegida.
- **distribucioGaussiana(int n), distribucioChi2(int n):** genera un array de dimensión n con valores que siguen una distribución gaussiana o chi cuadrado en el rango -1 a 1. Esto se consigue reescalando la distribución gausiana al dividir los valores obtenidos por el máximo en valor absoluto. Así nos aseguramos que todos los puntos estarán dentro de la ventana de trabajo y no habrá puntos fuera.
- **initSoluciones():** inicializa los atributos soluciones y distancias.

métodos públicos:

- **pushSolucion(Punto[] puntos):** comprueba si el punto forma parte de la solución y lo añade de ser así.
- **reset(Distribution distribution, int n, int nSolutions, Method typeSolution, String proximity):** este método permite reiniciar las variables y volver a generar los datos.
- **Getters y setters.**

IX. Implementación de la Vista

Para la implementación de la vista usaremos 5 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 puntos que contendrá la pantalla.

3. LeftLateralPanel.java: JPanel que aparecerá a la izquierda de la interfaz y que contendrá un conjunto de opciones que permitirán al usuario establecer las condiciones sobre las que se ejecutará el programa.

4. RightLateralPanel.java: JPanel que aparecerá a la derecha de la interfaz y que contendrá el botón de Start para arrancar la ejecución del programa, así como un Label en el que se mostrará el tiempo de ejecución del algoritmo una vez finalizada la ejecución del mismo, y otro Label que contendrá las los puntos que forman parte de las soluciones encontradas y las distancias asociadas.

5. Notificacion.java: JDialog que permite emitir mensajes temporales que dan feedback al usuario para facilitarle la interacción con el programa y prevenir errores.

View.java

Esta clase será una extensión de un JFrame en el que estarán ubicados todos los componentes de la visualización y será la interfaz entre la vista y el resto de componentes del programa.

Las variables que tendrá esta clase son:

- **GraphWidth** y **wiGraphHeight**: indican el ancho y la altura del GraphPanel que contendrá el JFrame.
- **MARGENLAT**: variable tipo int establece el margen de ancho en ambos lados que se deja para implementar los paneles laterales.
- **MARGENVER**: variable tipo int que establece un margen vertical.
- **graphPanel**: puntero hacia la clase GraphPanel.
- **leftPanel** : puntero a la clases LeftLateralPanel.
- **rightPanel** : puntero a la clasesRightLateralPanel.

- **controlador**: puntero al controlador.
- **modelo**: puntero al modelo.

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

métodos públicos:

- **mostrar():** inicializa la ventana principal y añade todos los elementos al JFrame.
- **paintGraph():** función que realiza un repaint sobre GraphPanel, es usada tanto por la propia clase View.java como por Controller.java
- **setTime(long nanoseconds):** Controller.java hace uso de esta función para establecer el valor de el long recibido por parámetro dentro del Label "Tiempo de ejecución (ns)" del RightPanel.
- **setBestResult():** función que utiliza Controller.java para establecer dentro de el Label "Distancias solución/es" de RightPanel, los datos obtenidos al finalizar el algoritmo respecto a los puntos y sus distancias.
- **setControlador(Controller controlador):** setter utilizado por AlgoritmosAvanzadosP3.java durante la inicialización del programa para hacer un set del controlador.
- **getModelo():** getter que usa el main durante la inicialización del programa para hacer un set del controlador.
- **setModelo(Model modelo):** el main del programa hace uso de este setter para inicializar el modelo.
- **getGraphWidth():** función usada tanto por el controlador como por el modelo y que devuelve el Width del GraphPanel.
- **getGraphHeight():** función usada tanto por el controlador como por el modelo y que devuelve el Height del GraphPanel.

métodos privados:

- **getCaptura(JPanel panel, int x1, int y1, int x2, int y2):** función que permite realizar un screenshot de la zona definida por los int que se pasan por parámetro.

métodos protegidos:

- **generatePointsClicked():** método que se utiliza desde el LeftLateralPanel para generar el conjunto de puntos que forman el modelo.
- **startClicked():** función que se utiliza desde el RightLateralPanel para arrancar la ejecución del controlador.

métodos sobrescritos:

- **mouseClicked(MouseEvent e):** con esta función se obtienen las coordenadas en las que se ha clicado con el objetivo de realizar ahi mismo la captura de panlla que se utilizará para visualizar el Zoom.

LeftLateralPanel.java

Esta clase será una extensión de un JPanel en el que se podrán visualizar todos los elementos que formarán el conjunto de opciones disponibles del panel lateral izquierdo del programa.

Las variables que tendrá esta clase son:

- **vista:** puntero a la vista.
- **distribution:** variable JComboBox que contendrá el conjunto de distintas distribuciones disponibles para nuestro programa.
- **proximity:** JComboBox que contiene las opciones que permiten seleccionar si lo que se buscará son los puntos más lejanos o los puntos más cercanos.
- **quantityPairs:** JComboBox que permite escoger cuantos puntos se van a crear.
- **solution:** JComboBox desde el cual el usuario podrá escoger entre realizar el algoritmo usando fuerza bruta o un divide and conquer.
- **quantityPoints:** JComboBox que permite escoger cuantos puntos se van a crear.
- **generateB:** variable tipo JButton usada como botón que se encarga de crear los puntos que forman el modelo de datos.
- **x,y:** variables tipo int que determinan el punto en el que se posiciona el JPanel
- **width, height:** variables tipo int que determinan el ancho y alto respectivamente del JPanel.

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

métodos privados:

- **init():** esta función es la encargada de inicializar todos los componentes que forman parte del panel lateral izquierdo del programa.

métodos protegidos:

- **getDistribution():** getter típico, en este caso usado por la clase View para obtener el tipo de distribución seleccionada por el usuario.
- **getProximity():** getter que utiliza la vista para obtener la selección del usuario en cuanto a la proximidad con la que ejecutar el algoritmo.
- **getQuantityPairs():** getter que utiliza la vista para obtener la selección del usuario en cuanto a la cantidad de parejas de puntos que buscará el algoritmo como solución.
- **getQuantityPoints():** getter que utiliza la vista para obtener la selección del usuario en cuanto a la cantidad de puntos con la que ejecutar el algoritmo.
- **getSolution():** getter que utiliza la vista para obtener el tipo de algoritmo con el que se quiere ejecutar el programa.

métodos sobrescritos:

- **actionPerformed(ActionEvent ae):** método asociado al JButton y que se encarga de llamar a la función generatePointsClicked() de la vista.

RightLateralPanel.java
Esta clase será una extensión de un JPanel en el que se podrán visualizar todos los elementos que formarán el conjunto de extras disponibles del panel lateral derecho del programa.

Las variables que tendrá esta clase son:
- **vista:** puntero a la vista.
- **x,y:** variables tipo int que determinan el punto en el que se posiciona el JPanel
- **width, height:** variables tipo int que determinan el ancho y alto respectivamente del JPanel.
- **startB:** variable tipo JButton usada como botón que se encarga de arrancar el controlador.
- **soluciones:** variable tipo JPanel en la que se mostrarán las soluciones encontradas al usuario una vez finalice la ejecución del programa.
- **timePanel:** variable tipo TimePanel que contendrá información respecto a los tiempos de ejecución de los algoritmos.

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

métodos públicos:

- **setTime(long nanoseconds):** función usada por la vista para establecer el tiempo de ejecución del algoritmo una vez finalizado.

métodos privados:

- **init():** esta función es la encargada de inicializar todos los componentes que forman parte del panel lateral izquierdo del programa.

métodos sobrescritos:

- **actionPerformed(ActionEvent ae):** método asociado al JButton y que se encarga de llamar a la función startClicked() de la vista.

Dentro de **RightLateralPanel** tenemos la clase anidada **TimePanel.** Esta clase no es más que un JPanel que contiene un JLabel y que permite mostrar al usuario los tiempos de ejecución del algoritmo.

Las variables que tendrá esta clase son:
- **timeLabel:** variable tipo JLabel que contendrá el String con el tiempo de ejecución en nanosegundos.

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

métodos públicos:

- **setTime(long nanoseconds):** función típica de setter, en este caso se usa desde la misma RightLateralPanel para hacer un set del tiempo de ejecución obtenido.

GraphPanel.java
Esta clase será una extensión de un JPanel que contendrá el conjunto de puntos con los que trabajará el programa.

Las variables que tendrá esta clase son:
- **vista:** puntero a la vista.

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

métodos protegidos:

- **paintComponent(Graphics g):** función que permite el pintado de los distintos puntos que forman el conjunto de datos del modelo.

Notificacion.java
La clase Notificación es una extensión de JDialog que se encargará de emitir mensajes en forma de ventana emergente temporal para dar feedback al usuario y facilitarle el uso del programa.

La única variable de esta clase es:
- **timer**: variable de tipo Timer que permite establecer la temporalidad de la ventana emergente.

método sobrescrito:

- **paint(Graphics g):** función que se encarga de la visualización por pantalla de la ventana de notificación.

XII. Estudios De Los Algoritmos

En este proyecto hemos desarrollado 4 tipos de Soluciones o algoritmos, los cuales hemos nombrado como: **Fuerza Bruta, Dividir y Vencer, Ávido y Heurístico.** A continuación explicaremos cada uno de los 4 tipos mencionados y estudiaremos como se comportan al variar la cantidad de puntos **N** como la **proximidad (**si tratamos de buscar las parejas más cercanas o las parejas más lejanas).

Conceptos previos:
Cuando nos referimos a encontrar la pareja de puntos más cercana o más lejana, lo que estamos haciendo en buscar la distancia mínima o máxima entre cualquier pareja de puntos. La distancia entre dos puntos se define como:

$$(distancia)_{p1p2} = \sqrt[2]{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Algoritmo "Fuerza Bruta"

Se trata de la primera aproximación, y la más simple algorítmicamente. Se trata de calcular la distancia de cada uno de los puntos con todos los demás. El algoritmo resultante presentaría una eficiencia computacional de **O(n²).**

Este algoritmo se podría optimizar parcialmente si únicamente comprobamos y formamos parejas no repetidas.
Si hablamos del número de combinaciones sin repetirse que calcularemos serán:

$$\frac{n!}{r!(n-r)!}$$

Para una N = 1.000.000, comprobaríamos el total de combinaciones posibles igual a 499.999.500.000 (por lo que se trataría de la mitad de combinaciones aproximadamente). Lo que nos sería inviable a partir de los 100.000 N (4.999.950.000 combinaciones), por lo que nos vemos obligados a buscar otras alternativas.

Algoritmo "Dividir y Vencer"

El algoritmo de Dividir y Vencer consiste en 3 pasos.
En primer lugar, ordenaremos todos los puntos según una de las coordenadas.
En segundo lugar, vamos dividiendo el array de puntos por la mitad utilizando los índices de la izquierda y la derecha hasta llegar a los casos base:

- Array de 1 elemento: retorna el valor máximo o mínimo de tipo Double en función de si se buscan los puntos más cercanos o más lejanos.
- Array de 2 elementos: retorna la distancia entre los dos puntos.
- Array de 3 elementos: retorna el mínimo o el máximo de las distancias entre las 3 parejas de puntos.
- Array de más de 3 elementos: divide el array por la mitad y hace la llamada recursiva de cada lado.

Finalmente, sacamos la distancia máxima entre la solución de la mitad izquierda y la derecha (distanciaPorcion), recorremos los puntos de la división actual y guardamos en el array llamado "porción" los puntos que están a menos de distanciaPorcion de distancia en el eje X. Luego ordenamos los puntos de la porción por la coordenada Y para evaluar solo los puntos que están a menos de esta distancia en el eje Y. Una vez que tenemos un candidato a solución, hacemos modelo.pushSolucion() con la pareja de puntos para ir almacenando las n mejores soluciones encontradas hasta el momento.
Este algoritmo tiene un coste asintótico de $O(n \cdot log(n))$, ya que dividimos el array por mitades y para cada una de las divisiones hacemos todos con todos de forma optimizada.
En adición, hemos hecho que guarde en una HashMap la frecuencia de aparición de cada tamaño de porción (cantidad de puntos con los que hace todos con todos) y la cantidad de puntos puede ir desde 2 puntos hasta 400 o más, por lo que coger 7 puntos no es suficiente cuando estamos trabajando con coordenadas de puntos en magnitud real.

Por último se encuentran los que hemos denominado Algoritmo **Ávido** y algoritmos **Heurísticos.** Estos algoritmos únicamente se centran en la búsqueda de la pareja de puntos más lejana, y sus implementaciones variarán según la distribución de Puntos que se trate.

Algoritmo Ávido en una Distribución Uniforme

Al trabajar con los otros 2 tipos de algoritmos en busca de las parejas más lejanas nos dimos cuenta de que la pareja de puntos siempre se encontrará en una combinación de puntos situados en esquinas opuestas. Este caso se convierte en más probable cuanto mayor es el tamaño de N y un espacio cuadrado o rectangular. (el cual sería nuestro caso). Pero intentando llegar a un razonamiento matemático, nos dimos cuenta de que se trataba de aquellas esquinas en cualquier polígono regular, es decir, aquellos que todos sus lados son de la misma longitud.

Dejando un lado el resto de polígonos, nos centraremos en nuestro caso. Un polígono regular de 4 lados (Cuadrado).

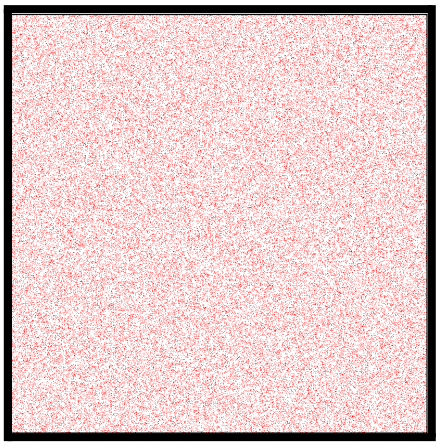

Fig. X: Nube de 100.000 puntos, distribución Uniforme.

Tal como se ha comentado, y como se puede deducir, la pareja más lejana será aquella formada por puntos situados en esquinas opuestas. Por lo que esta vez, en vez de ordenar los puntos por una de las coordenadas, los ordenaremos según la distancia que se encuentran del punto central del cuadrado.

Una vez que hayamos ordenado todos los puntos, recogeremos aquellos que se encuentran más alejados del centro, más específicamente l**os** $\sqrt{n}$ **primeros (**Tal como hemos comentado en el apartado anterior de **Algoritmo "Dividir y Vencer"**).

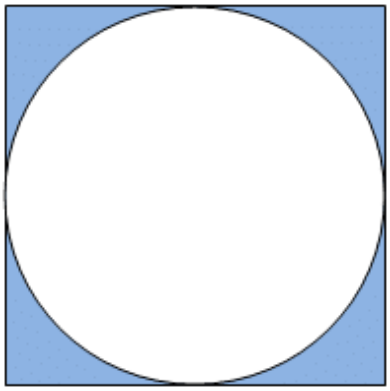

Fig. X: Representación $\sqrt{n}$ puntos. más alejados del centro en Algoritmo Ávido - Distribución Uniforme.

Una vez que tengamos l**os $\sqrt{n}$ primeros,** realizaremos todas las combinaciones posibles entre ellos (Fuerza bruta). Este fragmento del algoritmo se trataría de un O(n²), pero debido a que estamos trabajando con $\sqrt{n}$, por lo tanto, se convertiría en : $O((\sqrt{n})^2) => O(n)$ **Sin embargo, al tener que utilizar un método de ordenación, el algoritmo acaba por ser O(n·log(n))**

Algoritmo Ávido en una Distribución Normal (Gaussiana)
En el caso de una distribución Gaussiana, también podemos utilizar la misma estratégia de ordenación, selección y combinación que en la distribución uniforme.

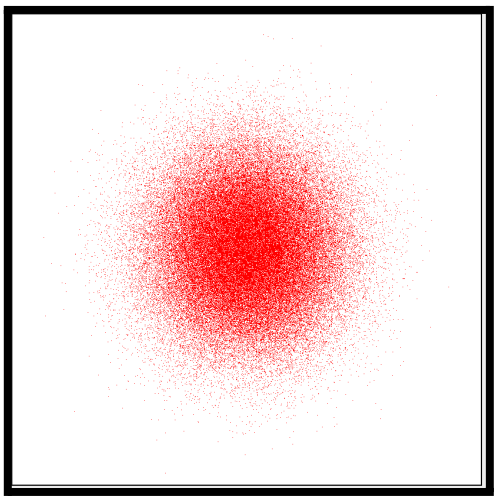

Fig. X: Nube de 100.000 puntos, distribución Gaussiana.

Tal como se puede observar en la figura anterior, la mayor parte de los puntos se encontrarán en la zona más cercana al punto central. Por lo que realizando los mismos pasos ya explicados (Ordenación según la distancia al Punto central y selección de los $\sqrt{n}$ más lejanos), estos puntos estarán situados en el area azul de la siguiente figura:

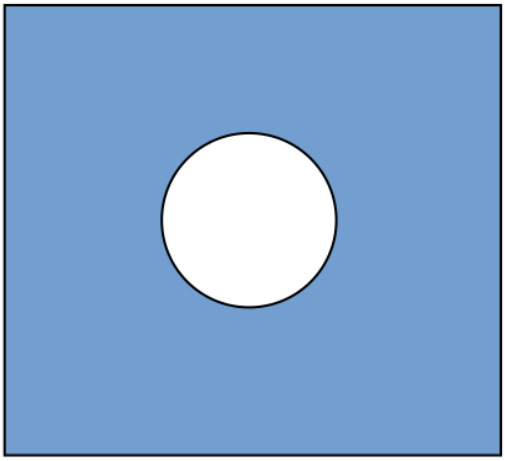

Fig. X: Representación $\sqrt{n}$ puntos. más alejados del centro en Algoritmo Ávido - Distribución Gaussiana

Una vez que tengamos l**os $\sqrt{n}$ primeros,** realizaremos todas las combinaciones posibles entre ellos (Fuerza bruta). Este fragmento del algoritmo se trataría de un O(n²), pero debido a que estamos trabajando con $\sqrt{n}$, por lo tanto, se convertiría en : $O((\sqrt{n})^2) => O(n)$. Sin embargo, al tener que utilizar un método de ordenación, el algoritmo acaba por ser $O(n \cdot log(n))$. **Igual manera que se comporta el algoritmo Ávido para la distribución uniforme**

Algoritmo Ávido en una Distribución Chi Cuadrado
En el caso de una distribución Chi², también podemos utilizar la misma estrategia de ordenación, selección y combinación que en la distribución uniforme y gaussiana.

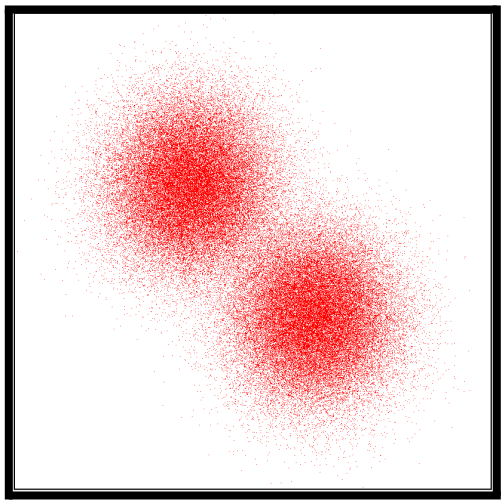

Fig. X: Nube de 100.000 puntos, distribución Chi².

A diferencia de la distribución gaussiana, la mayor parte de los puntos se encontrará dividida en las 2 secciones circulares situadas literalmente desplazadas del centro. Debido a que ambas secciones se encuentran separadas por la misma distancia del punto central del mapa, si obtenemos el punto

medio a partir de ambos puntos centrales, obtendremos nuevamente el punto central como punto de referencia. Por lo que podremos realizar los mimos pasos ya explicados (Ordenación según la distancia al Punto central y selección de los $\sqrt{n}$ más lejanos), estos puntos estarán situados en el área azul de la *Figura X (Representación $\sqrt{n}$ puntos más alejados del centro en Algoritmo Ávido - Distribución Gaussiana)*.

Una vez que tengamos los n primeros, realizaremos todas las combinaciones posibles entre ellos (Fuerza bruta). Este fragmento del algoritmo se trataría de un O(n²), pero debido a que estamos trabajando con n , por lo tanto, se convertiría en : O((n)²) => O(n). Sin embargo, al tener que utilizar un método de ordenación, el algoritmo acaba por ser O(n·log(n))

Conclusión Algoritmos Ávidos:
La principal función de los algoritmos ávidos mencionados consiste en la reducción de parejas de puntos que se deben combinar y comparar para encontrar la mejor solución.

Esta función la cumple de una manera bastante fiable y sencilla, el único problema es la ordenación del array de Puntos. Esta ordenación es el mayor cuello de botella del algoritmo, pero en comparación con los 2 otros tipos de algoritmos mencionados (Fuerza bruta y Dividir y Vencer), a partir de un N > 1000 se puede confirmar con bastante firmeza que es el mejor algoritmo para encontrar la pareja de puntos más lejana, ya se trate de cualquiera de las 3 distribuciones comentadas. A partir de estas soluciones ávidas comentadas, surge la idea de, "¿Hay alguna otra forma de obtener esos $\sqrt{n}$ puntos sin necesidad de realizar una ordenación (n*log(n)) y se mantenga un coste computacional igual o menor?". De ahí surgen los algoritmos que denominamos como **Algoritmos Heurísticos.**

Algoritmos Heurísticos
Como acabamos de comentar estos algoritmos surgen de la idea de obtener una cantidad aproximada de los $\sqrt{n}$ puntos que obtendremos realizando el Algoritmo Ávido sin realizar la ordenación del array de puntos (n*log(n)).

Los algoritmos que trataremos a continuación son soluciones específicas a las 3 distribuciones tratadas en este artículo (Uniforme, Gaussiana y Chi²), por lo que su uso en cualquier otra distribución no daría resultados fiables. Además, también se tratan de espacios Cuadrados finitos, por lo que podemos saber y calcular los puntos de referencia para estos algoritmos heurísticos.

Algoritmo Heurístico en Distribución Uniforme
Al haber realizado el resto de algoritmos suficientes veces podremos comprobar que para la distribución Uniforme, la pareja de puntos más alejada será aquella formada por puntos situados en esquinas opuestas. Y cuanto mayor es N, más probable es. A partir de una densidad del 50%, podemos confirmar que la pareja se encontrarán en esquinas opuestas con un 100% de certeza, por lo que este algoritmo se centrará en despreciar aquellos puntos no cercanos a las esquinas y únicamente formar parejas con aquellos situados a una distancia suficiente para obtener una cantidad total de $\sqrt{n}$ puntos o mayor.

El algoritmo consiste en un recorrido por todos los puntos O(n), por el cual, dependiendo de que esquina se trate (Superior Izquierda, Superior Derecha, Inferior Izquierda o Inferior Derecha) se calculará la distancia del punto a su esquina correspondiente, los cuales sus coordenadas serán: (0,0) ; (Width, 0) ; (0,Heigh)t ; (Width, Height), siendo Width y Height el Ancho y el Alto del Mapa.

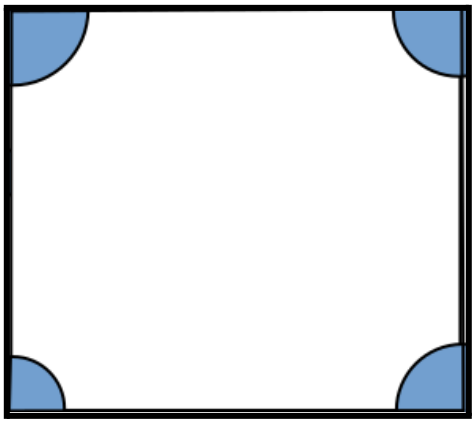

Fig. X: Representación $\sqrt{n}$ puntos en Algoritmo Heurístico - Distribución Uniforme

El radio de las circunferencias situadas en las esquinas se explicará más adelante de esta sección.

Una vez que hemos obtenido los puntos de cada sección, Realizaremos fuerza bruta (O(n²)) para calcular todas las combinaciones posibles y.así obtener la pareja (o parejas) más lejanas.

Al tratarse de una cantidad de n muy reducida ($\sqrt{n}$), no tendrá coste (O(n²)), sino que se convertirá en $O((\sqrt{n})^2) => O(n)$.

Por lo tanto, el coste computacional de este Algoritmo será O(2n) => O(n), Convirtiéndose a primera vista en una opción mejor que el Algoritmo Ávido y así, la mejor opción de los 4 distintos tipos de Algoritmos.

La distancia que para obtener los $\sqrt{n}$ puntos se dejó como fija, invariable del tamaño N, por lo que a medida que aumenta N, el algoritmo Ávido se convierte en una solución más fiable. Pero eso no significa que no se podría dejar como una distancia variable.

Tal como se ha mencionado, a partir de una densidad determinada sabemos que la solución se encontrará en una combinación de puntos situados en esquinas opuestas, por lo que también podríamos usar la densidad para determinar cuál será la distancia con el que delimitaremos el área de cada esquina. **(Únicamente para la distribución Uniforme en un mapa Finito)**

Pongamos que el ancho y el alto del mapa es 500. Para un N = 100.000, tendríamos 500² / 100.000, lo que saldría a 1 punto por cada 2.5. Si obtenemos este valor, lo multiplicamos por un valor fijo (llamémosle K) y determinamos la distancia de una esquina (0,0 por ejemplo) a un punto situado 2'5 * K (siendo 3 por ejemplo -> el punto (7,5 ; 7,5) podríamos obtener una distancia variable y una cantidad de $\sqrt{n}$ puntos dependiendo del tamaño de N, por lo que su eficiencia no variaría, su fiabilidad no se vería afectada (Ya que obtendríamos una cantidad de $\sqrt{n}$ puntos o más) y seguiría siendo un algoritmo mejor (Computacionalmente) que el Algoritmo Ávido.

Desgraciadamente debido a la falta de tiempo no pudimos obtener este valor K, pero de esta forma dejamos constancia como quedaría optimizado.

Algoritmo Heurístico en Distribución Gaussiana

Cambiando ahora a la Distribución Gaussiana, partiremos del mismo objetivo de obtener esos $\sqrt{n}$ puntos sin la ordenación (que conllevaría un O(n*log(n)). Para ello, decidimos en este caso realizar un recorrido de todos los N puntos (O(n)) y quedarnos en este caso aquellos situados a una distancia D del punto Central. Esta distancia D sería la suficiente como para que una vez recorrido todos los puntos obtengamos una cantidad igual o mayor de $\sqrt{n}$ puntos (Siempre será mayor).

Para la representación de los $\sqrt{n}$ puntos, se trata de la misma figura, *Fig. X: Representación $\sqrt{n}$ puntos. más alejados del centro en Algoritmo Ávido - Distribución Gaussiana*

Una vez obtenidos realizaremos fuerza bruta para calcular todas las combinaciones y obtener todas las soluciones que se requieran. Y tal como se explicó en la sección anterior, el coste computacional de esta sección será de $O((\sqrt{n})^2) => O(n)$. Y Por lo tanto, el coste computacional de este Algoritmo será O(2n) => O(n).

En esta distribución, se podría mirar que cambiar la Distancia Fija D y hacerla dinámica según el tamaño de N, pero para tamaños hasta N < 5.000.000, este algoritmo sigue siendo mejor que el Ávido para esta distribución, por lo que (aparte de la falta de tiempo) decidimos tanto este como la siguiente distribución darla por finalizada (no realizar más mejores u optimizaciones).

Algoritmo Heurístico en Distribución Chi²

Por último, queda la Distribución Chi². El Algoritmo Heurístico de esta distribución, parte de una idea similar al comentado par a la distribución Gaussiana. Para querer eliminar la mayor parte de los puntos innecesarios querremos eliminar aquellos situados a los puntos centrales de las 2 áreas de los círculos de la distribución.

En primer lugar, realizaremos un recorrido de todos los N puntos (O(n)) y nos quedaremos en este caso aquellos situados a una distancia D del punto central de las 2 circunferencias, es decir, calcularemos la distancia del punto p, a los dos puntos centrales p1 y p2 de las circunferencias de la distribución Chi², y en caso de que se cumpla la condición de no encontrarse cerca de ninguno de esos puntos (Una distancia mayor a D) entonces lo almacenaremos en un array auxiliar. Esta distancia D sería la suficiente como para que una vez recorrido todos los puntos obtengamos una cantidad igual o mayor de $\sqrt{n}$ puntos (Siempre será mayor).

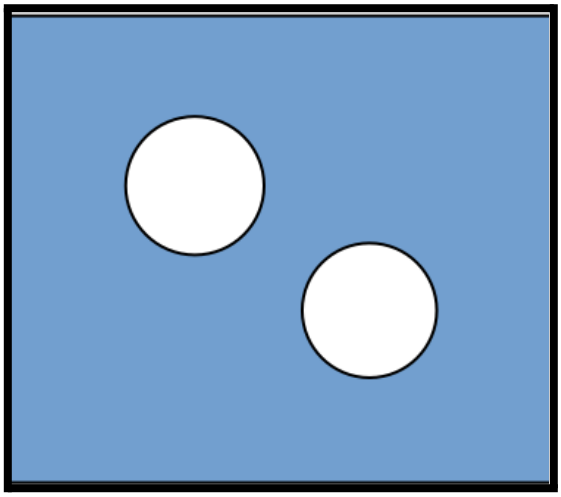

Fig. X: Representación $\sqrt{n}$ puntos en Algoritmo Heurístico - Distribución Chi²

Por último realizaremos fuerza bruta para calcular todas las combinaciones y obtener todas las soluciones que se requieran. Y tal como se explicó en la sección anterior, el coste computacional de esta sección será de $O((\sqrt{n})^2) => O(n)$. Y Por lo tanto, el coste computacional de este Algoritmo será O(2n) => O(n).

Conclusión Algoritmos Heurísticos:

Tal como hemos podido observar, podemos concluir que los algoritmos heurísticos, aunque no se basen en una base estadística demostrada como los algoritmos ávidos, sí que tratamos con una cantidad de puntos mayor que $\sqrt{n}$ por lo que debería tratarse de un algoritmo fiable (al menos lo es con nuestras distribuciones tratadas en el artículo). Sí que cabe

comentar que para tamaños N > 100.000 los algoritmos pierden fiabilidad y la mejor opción es el algoritmo ávido o el algoritmo de fuerza bruta.

XIV. 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. El **ratio** de la CPU de la máquina con la que se ha grabado el vídeo es de **0,9**.

XV. 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

En la interfaz de nuestra aplicación tenemos los siguientes componentes:
Panel izquierdo:

- **Tipo de distribución**: podemos seleccionar la distribución de los puntos en el plano 2D. Permite escoger entre una distribución de puntos aleatoria uniforme, gausiana y chi cuadrado. Se ha aplicado un reescalado para asegurar que todos los puntos generados estén dentro de la ventana de trabajo.
- **Proximidad:** el programa permite al usuario calcular los puntos más lejanos o los puntos más cercanos.
- **Nº de Parejas:** el programa permite calcular hasta las 10 parejas de puntos más cercanos o lejanos.
- **Tipo de solución:** Permite escoger el algoritmo para resolver el problema. Se puede utilizar un algoritmo de fuerza bruta con un coste de complejidad $O(n^2)$ o bien un algoritmo del tipo dividir y vencer con un coste $O(n \cdot \log n)$. Para el cálculo de las parejas lejanas en un espacio equiprobable (distribución uniforme) se ha implementado un algoritmo ávido.
- **Cantidad de puntos:** El programa permite al usuario escoger la cantidad de puntos a generar. Se puede escoger entre 1.000, 10.000, 100.000 y 1000.000.
- Botón **Generar Puntos:** al pulsarlo el programa genera la nube de puntos 2D con la cantidad de puntos previamente seleccionada y según la distribución indicada por el usuario.

Panel derecho:

- **Tiempo de Ejecución (ns):** tras la ejecución del programa se muestra el tiempo que ha tardado en nanosegundos.
- **Distancias soluciones:** en este cuadro de texto se muestran las coordenadas de las parejas de puntos que son solución y la distancia entre ellos.
- Botón **Start:** ejecuta el programa.

XVI. Conclusión

Esta práctica nos ha permitido experimentar y realizar una implementación de un programa usando MVC y profundizar en el conocimiento de los algoritmos de dividir y vencer.

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.