Universidad de Huelva

ESTRATEGIAS ALGORÍTMICAS

Práctica 1 de AMC

ANTONIO ABAD HERNÁNDEZ GÁLVEZ
3º GRADO DE INGENIERÍA INFORMÁTICA

ETSI
ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA
uhu.es

ÍNDICE DE CONTENIDOS

1.- Introducción

El problema del par de puntos más cercano.

El problema del punto más cercano es un problema clásico en geometría computacional. Dado un conjunto de *n* puntos en el plano, el objetivo es encontrar el par de puntos más cercano, es decir, los 2 puntos cuya distancia entre sí sea la mínima de entre todos los que hay en el plano. La distancia entre dos puntos **p**=(x1, y1) y **q**=(x2, y2) en el plano cartesiano se define matemáticamente por la fórmula euclidiana:

$$d(p,q) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Este problema tiene aplicaciones en varias áreas, como la informática (por ejemplo, en redes inalámbricas para encontrar dispositivos cercanos) y la astronomía (para identificar objetos celestes próximos entre sí).

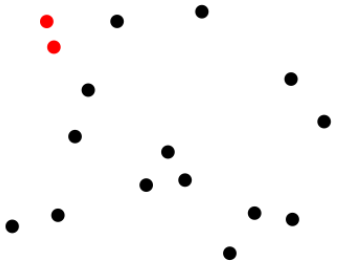


Imagen 1. Un plano con un conjunto de puntos donde el par de puntos más cercano está en rojo.

2. Estrategias de Aproximación a la Solución del Problema Planteado

Para resolver este problema, existen varias estrategias de búsqueda, aunque nosotros nos centraremos en las 3 siguientes, cada una con diferentes niveles de complejidad y eficiencia:

1. Estrategia basada en Algoritmos Exhaustivos

La estrategia de algoritmos exhaustivos (o fuerza bruta) consiste en evaluar todos los pares posibles de puntos en el conjunto para determinar cuál de ellos tiene la distancia mínima. Dado un conjunto de *n* puntos, el número de pares posibles es $\binom{n}{2} = \frac{n\,(n-1)}{2}$, por lo que la complejidad temporal de este enfoque es O(n²).

Proceso:

- Se calcula la distancia entre cada par de puntos.
- Se almacena la distancia mínima encontrada hasta el momento.
- Se compara la distancia mínima actual con la distancia de cada par de puntos.
- Si la distancia mínima es menor que la actual, actualizamos el valor de la distancia mínima. Si no lo es, seguimos comparando.
- Al final, se devuelve el par de puntos con la distancia mínima.

Ventajas:

- Es fácil de implementar (corto y lógica sencilla).
- Es muy rápido para conjuntos pequeños de puntos, ya que no necesita optimización específica.

Desventajas:

- Se vuelve ineficiente a medida que el número de puntos aumenta debido a su complejidad cuadrática.

2.- Estrategia basada en Algoritmos Exhaustivos con Poda

La estrategia de algoritmos exhaustivos con poda optimiza el enfoque de fuerza bruta mediante la reducción del número de pares de puntos que se evalúan. La poda consiste en descartar ciertos pares de puntos en función de reglas predefinidas, sin necesidad de calcular la distancia entre ellos explícitamente.

Proceso:

- Ordenar los puntos de acuerdo a una coordenada, lo que permite analizar zonas de forma más organizada.
- Utilizar condiciones de poda para ignorar pares que se encuentran demasiado lejos en una coordenada (por ejemplo, ignorando pares de puntos si la diferencia en una de sus coordenadas ya es mayor que la distancia mínima encontrada).
- Evaluar solo los puntos en una región de interés, reduciendo así el número de comparaciones.

Ventajas:

- Reduce la cantidad de comparaciones necesarias y, por lo tanto, puede mejorar la eficiencia en casos específicos.

Desventajas:

- La poda puede no reducir suficientemente el número de comparaciones en todos los casos, y en algunos casos puede ser tan lenta como el método de fuerza bruta.
- Puede requerir una estructura de datos adicional para organizar los puntos, lo cual aumenta la complejidad de la implementación.
- Se vuelve ineficiente a medida que aumentamos la talla del vector de puntos debido a su complejidad cuadrática.

3. Estrategia basada en Algoritmos Divide y Vencerás

Los algoritmos de Divide y Vencerás abordan el problema de una manera más eficiente dividiendo el conjunto de puntos en subproblemas más pequeños, resolviendo cada uno de ellos y combinando los resultados. Este enfoque mejora considerablemente la eficiencia, logrando una complejidad de **O(n log n)**, que es una mejora significativa sobre el enfoque de fuerza bruta.

Proceso:

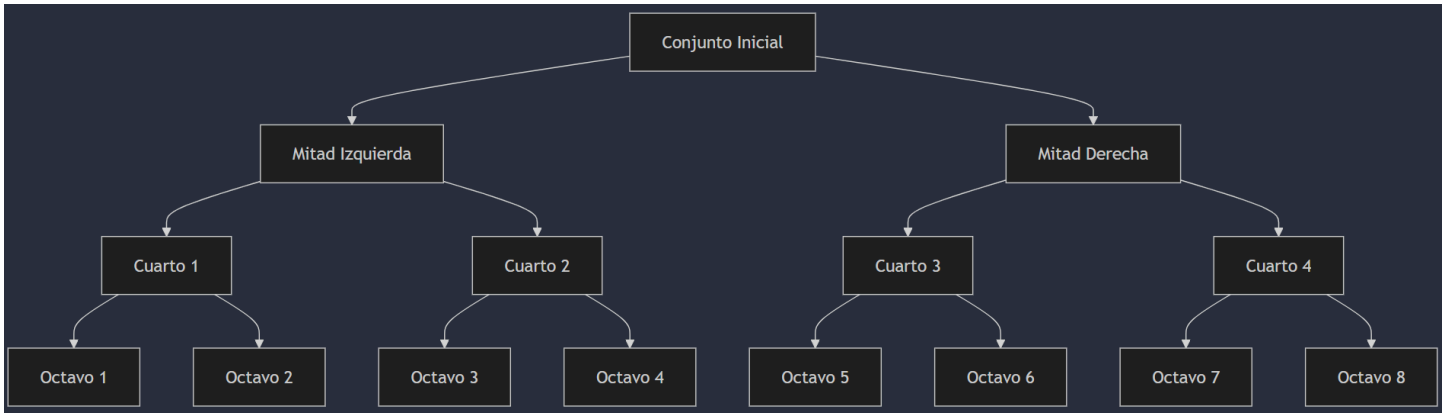
- Se ordenan los puntos según sus coordenadas x.
- El conjunto de puntos se divide recursivamente en dos subconjuntos aproximadamente iguales, aplicando el mismo algoritmo de forma recursiva en cada subconjunto.
- Una vez que se ha encontrado la distancia mínima en cada subconjunto, se combinan los resultados. Para esto, se analiza la zona cercana a la línea divisoria (banda de ancho $2 \cdot \delta$, siendo δ la mínima distancia encontrada en los subconjuntos), comparando únicamente los puntos que están dentro de esta región y que podrían tener una distancia menor.

Ventajas:

- Mucho más eficiente para conjuntos grandes, reduciendo el tiempo de ejecución a **O(n log n)**.
- Permite resolver problemas más grandes que serían imposibles de manejar con un enfoque exhaustivo simple.

Desventajas:

- La implementación es más compleja, ya que requiere ordenar los puntos y manejar recursivamente los subproblemas.
- Requiere una estrategia de combinación eficiente en la fase de "conquista" del algoritmo.



3. Solución particular planteada

Para el cálculo de las distancias en cada algoritmo hemos utilizado el método 'calcularDistancia(a, b)', cuyo pseudocódigo es el siguiente:

funcion **calcularDistancia**(a: Punto, b: Punto) return double
    d=raiz_cuadrada(b.x-a.x)^2+(b.y-a.y)^2;
    return d;
ffuncion

1.- Algoritmo Exhaustivo

A continuación, dejamos la solución **Exhaustiva** en código **JAVA** y su **pseudocódigo**

JAVA

```java
 * @param v_puntos Es un vector de tipo 'Punto' que contiene todos los Puntos que han sido cargados desde el fichero
 * @param izq Es la posicion inicial (mas a la izquierda) del vector 'v_puntos'
 * @param dch Es la posicion final (mas a la derecha) del vector 'v_puntos'
 * @return Devuelve un vector 'sol' de tipo Punto con los 2 Puntos mas cercanos de todos los que hay cargados en el vector 'v_puntos'
 */
public static Punto[] busquedaExhaustiva(Punto v_puntos[], int izq, int dch)
{
    double dis; // Distancia utilizada para comparar la distancia entre 2 Puntos en cada vuelta de bucle
    Punto sol[]=new Punto[2]; // Vector solucion en el que cargaremos los 2 Puntos mas cercanos

    // Inicialmente, empezamos a recorrer el Vector 'v_puntos' de izq a dch, y por tanto, los primeros puntos candidatos a ser la solucion son los 2 primeros
    sol[0]=v_puntos[0];
    sol[1]=v_puntos[1];

    // Recorremos el vector 'v_puntos' y vamos comparando la distancia de los Puntos
    for(int i=izq; i<=dch; i++)
    {
        /* Los Puntos anteriores a 'i' NO debemos recorrerlos, ya que sino el Algoritmo seria menos eficiente porque estariamos comparando un Punto que ya ha sido comparado
        anteriormente, es decir, si hemos comparado P(2,3) NO debemos comparar P(3,2) porque ya conocemos su distancia */
        for(int j=i+1; j<=dch; j++)
        {
            // Comparamos la distancia de los 2 Puntos adyacentes en el fichero
            dis=Algoritmo.calcularDistancia(v_puntos[i], v_puntos[j]);
            if(dis < Algoritmo.dis_min) // Si la distancia entre esos 2 puntos es menor que la distancia minima, realizamos un swap
            {
                Algoritmo.dis_min=dis;
                sol[0]=v_puntos[i];
                sol[1]=v_puntos[j];
            }
        }
    }
    return sol;
}
```

Pseudocódigo

funcion **busquedaExhaustiva**(v_puntos: Punto[], izq: int, dch: int) return Punto[]

```
1   sol[0]=v_puntos[0];
2    sol[1]=v_puntos[1];
3     para(i=izq hasta dch con incremento 1)
4           para(j=i+1 hasta dch con incremento 1)
5                 dis=calcularDistancia(v_puntos[i], v_puntos[j]);
6                si(dis < dis_min) entonces
7                       dis_min=dis;
8                       sol[0]=v_puntos[i];
9                       sol[1]=v_puntos[j];
10               fsi
11        fpara (j)
12    fpara (i)
13    return sol;
```

ffuncion

1.1. Caso Mejor

El caso mejor se da cuando el par de puntos que tiene la distancia mínima está formado por los puntos del vector de las posiciones 0 y 1 respectivamente, es decir, cuando el vector está ordenado de menor a mayor. De esta manera, no entra nunca en el ***if*** de la línea 6.

Sin embargo, aunque no entre en el ***if*** siempre realiza todas las iteraciones de los dos bucles ***for***, ya que no hay ninguna condición que pare dichos bucles.

Por lo tanto, el orden de complejidad del caso mejor es de $\mathbf{O(n^2)}$.

1.2. Caso Peor

El caso peor se da cuando el par de puntos que tiene la distancia mínima está formado por los puntos del vector de las dos últimas posiciones, es decir, cuando el vector está ordenado de mayor a menor, ya que cada vez que calculamos la distancia entre 2 puntos en cada vuelta de bucle, obtenemos una distancia mayor, y por tanto, podemos decir que el vector está ordenado de mayor a menor, porque la distancia entre los puntos 0 y 1, es mayor que la distancia entre los puntos 0 y 2, y así sucesivamente.

De esta manera, siempre entra en el ***if*** de la línea 6.

Además de entrar en el ***if*** siempre realiza todas las iteraciones de los dos bucles ***for***, ya que no hay ninguna condición que pare dichos bucles.
Por lo tanto, el orden de complejidad del caso peor es de $\mathbf{O(n^2)}$.

1.3. Caso Medio

El orden de complejidad del caso medio es el mismo que el del caso peor, es decir, $\mathbf{O(n^2)}$.

1.4. Gráficas

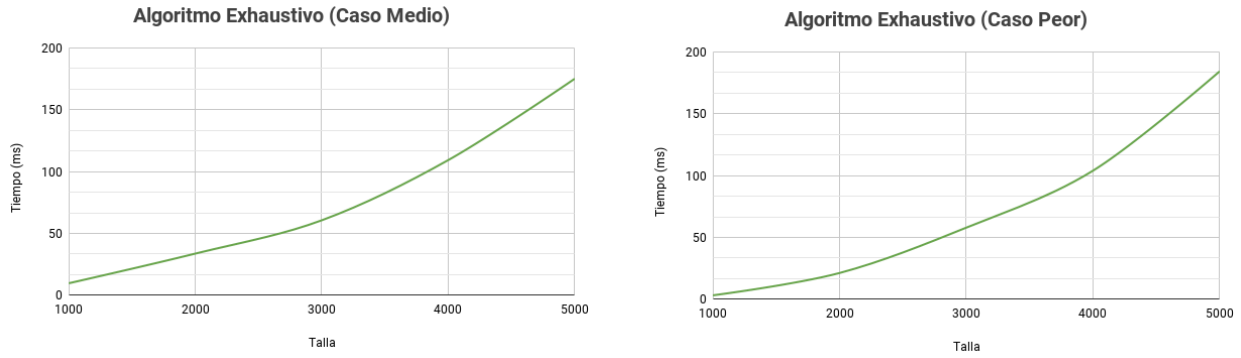


Imagen 2. Gráficas del Caso Medio y Caso Peor para el Algoritmo Exhaustivo.

2.- Algoritmo Exhaustivo con Poda

La poda consiste en no seguir buscando más puntos cuando la distancia de la coordenada **X** entre dos puntos sea mayor que la distancia mínima que tenemos calculada.

Para poder utilizar la búsqueda exhaustiva con poda, previamente en el 'main()' debemos ordenar los puntos del vector por la coordenada **X** mediante un algoritmo de ordenación que tenga un orden de complejidad de **O(n log n)**.
En nuestro código utilizamos el algoritmo **QuickSort**.

A continuación, dejamos la solución **Exhaustiva con Poda** en código **JAVA** y su **pseudocódigo**

JAVA

```java
 * La Poda consiste en NO seguir buscando mas Puntos cuando la distancia respecto al eje X entre 2 Puntos sea mayor que la distancia minima que tenemos calculada
 * Para poder utilizar la busqueda Exhaustiva con Poda, previamente en el 'main()' debemos ordenar los Puntos del Vector respecto al Eje X (QuickSort)
 * @param v_puntos Es un vector de tipo 'Punto' que contiene todos los Puntos que han sido cargados desde el fichero
 * @param izq Es la posicion inicial (mas a la izquierda) del vector 'v_puntos'
 * @param dch Es la posicion final (mas a la derecha) del vector 'v_puntos'
 * @return Devuelve un vector 'sol' de tipo Punto con los 2 Puntos mas cercanos de todos los que hay cargados en el vector 'v_puntos'
 */
public static Punto[] busquedaExhaustivaPoda(Punto v_puntos[], int izq, int dch)
{
    double dis; // Distancia utilizada para comparar la distancia entre 2 Puntos en cada vuelta de bucle
    Punto sol[]=new Punto[2]; // Vector solucion en el que cargaremos los 2 Puntos mas cercanos

    // Inicialmente, empezamos a recorrer el Vector 'v_puntos' de izq a dch, y por tanto, los primeros puntos candidatos a ser la solucion son los 2 primeros
    sol[0]=v_puntos[0];
    sol[1]=v_puntos[1];

    // Recorremos el vector 'v_puntos' y vamos comparando la distancia de los Puntos
    for(int i=izq; i<=dch; i++)
    {
        // Mientras NO lleguemos al final del Vector y nos interese seguir comparando, seguimos comparando las distancias entre los 2 puntos del eje X
        for(int j=i+1; j<=dch; j++)
        {
            // Si la distancia que hay entre los 2 Puntos del Eje X es mayor que la distancia minima, NO nos interesa seguir comparando (break), debemos Podar
            dis=Math.abs(v_puntos[j].getX() - v_puntos[i].getX());
            if(dis >= Algoritmo.dis_min) break;
            else
            {
                // Comparamos la distancia de los 2 Puntos adyacentes en el fichero
                dis=Algoritmo.calcularDistancia(v_puntos[i], v_puntos[j]);
                if(dis<Algoritmo.dis_min) // Si la distancia entre esos 2 puntos es menor que la distancia minima, realizamos un swap
                {
                    Algoritmo.dis_min=dis;
                    sol[0]=v_puntos[i];
                    sol[1]=v_puntos[j];
                }
            }
        }
    }
    return sol;
}
```

Pseudocódigo

```
funcion busquedaExhaustivaPoda(v_puntos: Punto[], izq: int, dch: int) return Punto[]
1     sol[0]=v_puntos[0];
2     sol[1]=v_puntos[1];
3     para(i=izq hasta dch con incremento 1)
4        para(j=i+1 hasta dch con incremento 1)
5           dis=valor_absoluto(v_puntos[j].x - v_puntos[i].x);
6           si(dis >= dis_min) entonces
7              break;
8           sino
9              dis=calcularDistancia(v_puntos[i], v_puntos[j]);
10             si(dis<dis_min) entonces
11                dis_min=dis;
12                sol[0]=v_puntos[i];
13                sol[1]=v_puntos[j];
14             fsi
15          fsi
16       fpara (j)
17    fpara (i)
18    return sol;
ffuncion
```

Pseudocódigo (QuickSort y Partition)

```
funcion QuickSort(v_puntos: Punto[], izq: int, dch: int, eje: char)
1     m=0;
2     segun(eje)
3        caso 'x':
4           si(izq < dch) entonces
5              m=Partition(v_puntos, izq, dch, 'x');
6              QuickSort(v_puntos, izq, m, 'x');
7              QuickSort(v_puntos, m+1, dch,'x');
8           fsi
9        fcaso
10
11       caso 'y':
12          si(izq < dch) entonces
13             m=Partition(v_puntos, izq, dch, 'y');
14             QuickSort(v_puntos, izq, m, 'y');
15             QuickSort(v_puntos, m+1, dch,'y');
16          fsi
17       fcaso
18    fsegun
ffuncion

funcion Partition(v_puntos: Punto[], izq: int, dch: int, eje: char) return int
1     pivote = v_puntos[izq];
2     i = izq - 1;
3     j = dch + 1;
4     segun(eje)
5        caso 'x':
6           repetir
7              repetir
8                 j--;
9              hasta (v_puntos[j].x > pivote.x);
10             repetir
11                i++;
12             hasta (v_puntos[i].x < pivote.x);
13             si (i < j) entonces
14                aux = v_puntos[i];
15                v_puntos[i] = v_puntos[j];
16                v_puntos[j] = aux;
17             fsi
18          hasta (i < j);
19       fcaso
20       caso 'y':
21          repetir
22             repetir
23                j--;
24             hasta (v_puntos[j].y > pivote.y));
25             repetir
26                i++;
27             hasta (v_puntos[i].y < pivote.y);
28             si (i < j) entonces
29                aux = v_puntos[i];
30                v_puntos[i] = v_puntos[j];
31                v_puntos[j] = aux;
32             fsi
33          hasta (i < j);
34       fcaso
35    fsegun
36    return j;
ffuncion
```

2.1. Caso Mejor

El caso mejor se da cuando se realiza una poda dentro de la condición del segundo bucle ***for*** (línea 7) en la segunda iteración de dicho bucle. Como el vector está previamente ordenado por la coordenada X de los puntos, si la distancia entre v_puntos[i] y v_puntos[j] es mayor o igual que la distancia actual 'dis' calculada en la primera iteración del segundo bucle ***for*** (línea 4) no interesa seguir comparando, ya que las distancias de los siguientes puntos serán mayor que la distancia actual de nuestra variable 'dis'.

Como la ordenación previa del vector de puntos la realizamos mediante el algoritmo de ordenación **QuickSort**, cuyo orden de complejidad es de **O(n log n)** y el segundo bucle ***for*** (línea 4) sólo realiza 2 iteraciones, obtenemos un orden de complejidad muy cercano a **O(n log n)**.

2.2. Caso Peor

El caso peor se da cuando nunca se realiza la poda dentro de la condición del segundo bucle ***for*** (línea 7) en la segunda iteración de dicho bucle. Como el vector está previamente ordenado por la coordenada X de los puntos, si la distancia entre v_puntos[i] y v_puntos[j] es menor que la distancia actual 'dis' calculada en la primera iteración del segundo bucle ***for*** (línea 4) debemos seguir comparando y nunca se para, es decir, no realizamos la poda nunca.
Como no realizamos ninguna poda, a efectos prácticos es como si tuviéramos un algoritmo exhaustivo (sin poda) en el que recorremos los 2 bucles ***for*** completos, y por tanto, el orden de complejidad es de **O(n^2)**.

El tiempo de ejecución es el mismo que el del algoritmo exhaustivo, pero en este, además, hemos tenido que ordenar el vector previamente mediante el algoritmo de ordenación QuickSort que tiene un orden de complejidad de **O(n log n)**, y por tanto, podemos afirmar que para el caso peor, el algoritmo exhaustivo es más eficiente que el exhaustivo con poda, ya que no tiene que realizar ninguna ordenación previa del vector de puntos.

2.3. Caso Medio

El orden de complejidad del caso medio depende del número de veces que se realiza la poda.
Si realizamos muchas podas, el orden de complejidad se acercará al mejor caso, y por lo tanto será próximo a **O(n log n)**.
No obstante, si prácticamente no realizamos ninguna poda, el orden de complejidad se acercará al peor caso, y por lo tanto será próximo a **O(n^2)**.

2.4. Gráficas

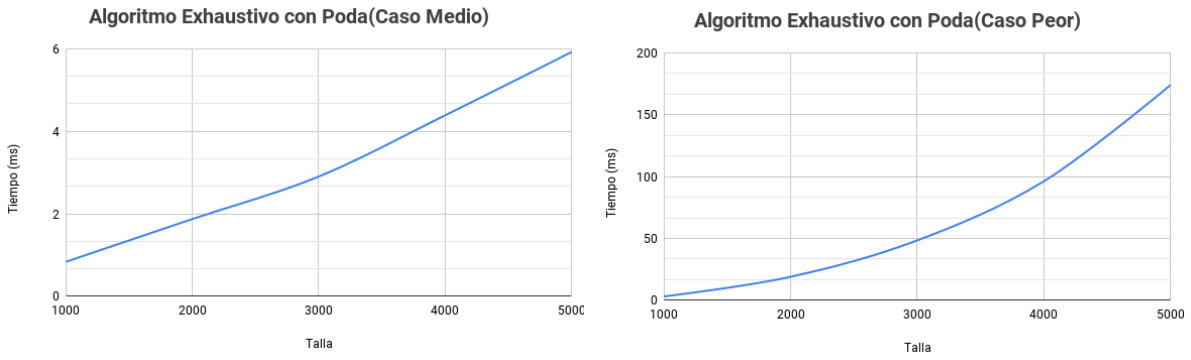


Imagen 3. Gráficas del Caso Medio y Caso Peor para el Algoritmo Exhaustivo con Poda.

3.- Algoritmo Divide y Vencerás

Para poder utilizar la búsqueda divide y vencerás, previamente en el 'main()' debemos ordenar los puntos del vector por la coordenada **X** mediante un algoritmo de ordenación que tenga un orden de complejidad de **O(n log n)**.
En nuestro código utilizamos el algoritmo **QuickSort**. (Pseudocódigo en 2.- Exhaustivo con Poda en la pág 10).

A continuación, dejamos la solución **Divide y Vencerás** en código JAVA y su pseudocódigo

JAVA

```java
/**
 * Para poder utilizar la busqueda mediante un Algoritmo basado en la tecnica Divide y Venceras, debemos ordenar los Puntos del Vector respecto al Eje X (QuickSort)
 * @param v_puntos Es un vector de tipo 'Punto' que contiene todos los Puntos que han sido cargados desde el fichero
 * @param izq Es la posicion inicial (mas a la izquierda) del vector 'v_puntos'
 * @param dch Es la posicion final (mas a la derecha) del vector 'v_puntos'
 * @return Devuelve un vector 'sol' de tipo Punto con los 2 Puntos mas cercanos de todos los que hay cargados en el vector 'v_puntos'
 */
public static Punto[] busquedaDyV(Punto v_puntos[], int izq, int dch)
{
    int nPuntos=dch-izq+1; // Nº de Puntos que hay en el Vector 'v_puntos'
    int mitad=izq+(dch-izq)/2; // Contiene la ultima posicion del Subvector de la Izquierda -> 0+(0+9)/2 = 4'5 = 4
    Punto sol_izq[]; // Vector solucion en el que cargaremos los 2 Puntos mas cercanos del Subvector de la izquierda
    Punto sol_dch[]; // Vector solucion en el que cargaremos los 2 Puntos mas cercanos del Subvector de la derecha
    Punto sol[]=new Punto[2]; // Vector solucion en el que cargaremos los 2 Puntos mas cercanos

    double dis_izq; // Distancia utilizada para comparar la distancia entre 2 Puntos del Subvector de la Izquierda en cada vuelta de bucle
    double dis_dch; // Distancia utilizada para comparar la distancia entre 2 Puntos del Subvector de la Derecha en cada vuelta de bucle
    double dis; // Distancia Actual calculada
    int i_izq=0; // Indice para recorrer el Subvector de la izquierda
    int i_dch=0; // Indice para recorrer el Subvector de la derecha

    if(nPuntos<4) // Si el vector 'v_puntos' tiene tantos Puntos como el Caso Base es mas eficiente utilizar el Algoritmo Exhaustivo con Poda
    {
        sol=Algoritmo.busquedaExhaustivaPoda(v_puntos, izq, dch);
    }
    else // Si el vector 'v_puntos' tiene mas Puntos que el Caso Base es mas eficiente utilizar la estrategia de Divide y Venceras
    {
        // Dividimos el Vector 'v_puntos' por la mitad en 2 Subvectores
        sol_izq=Algoritmo.busquedaDyV(v_puntos, izq, mitad);
        dis_izq=Algoritmo.dis_min;

        sol_dch=Algoritmo.busquedaDyV(v_puntos, mitad+1, dch);
        dis_dch=Algoritmo.dis_min;

        if(dis_izq<=dis_dch) // Si la distancia que hay entre los 2 Puntos del Subvector Izquierdo es menor que la distancia que hay entre los 2 Puntos del Subvector Derecho
        {
            Algoritmo.dis_min=dis_izq;
            sol=sol_izq;
        }
        else // Si la distancia que hay entre los 2 Puntos del Subvector Derecho es menor que la distancia que hay entre los 2 Puntos del Subvector Izquierdo
        {
            Algoritmo.dis_min=dis_dch;
        // Calculamos la distancia minima entre los Puntos que quedan en el Subvector Izquierdo
        for(i_izq=mitad; i_izq>=izq; i_izq--)
        {
            // Como el Vector inicial esta ordenado, si llegamos a un valor de 'i' en el que la distancia es mayor que la minima, los siguientes valores del Subvector tambien tendran una di
            dis=Math.abs(v_puntos[mitad].getX()-v_puntos[i_izq].getX());
            if(dis > Algoritmo.dis_min) break;
        }

        // Calculamos la distancia minima entre los Puntos que quedan en el Subvector Derecho
        for(i_dch=mitad+1; i_dch<=dch; i_dch++)
        {
            // Como el Vector inicial esta ordenado, si llegamos a un valor de 'i' en el que la distancia es mayor que la minima, los siguientes valores del Subvector tambien tendran una di
            dis=Math.abs(v_puntos[i_dch].getX()-v_puntos[mitad].getX());
            if(dis > Algoritmo.dis_min) break;
        }

        // Como la distancia minima entre 2 puntos puede estar entre un Punto de la Franja Izquierda y un Punto de la Franja Derecha
        // Recorremos el vector 'v_puntos' hasta las posiciones podadas y comparamos la distancia de los Puntos del Subvector de la Izquierda (i) y del Subvector de la Derecha (j)
        // En el caso de que la distancia minima se repita, la distancia minima sera la del primer Punto del Vector, si NO queremos que sea asi -> if(dis >= dis_min)
        for(int i=mitad; i>i_izq; i--) // Recorremos el Subvector de la Izquierda
        {
            for(int j=mitad+1; j<i_dch; j++) // Recorremos el Subvector de la Derecha
            {
                dis=(Math.abs(v_puntos[j].getX() - v_puntos[i].getX()));
                // Si la distancia que hay entre un Punto del subvector de la izquierda y un punto del subvector de la derecha es mayor que la distancia minima, NO debemos seguir comparando
                if(dis > Algoritmo.dis_min) break;
                else
                {
                    // Comparamos la distancia de los 2 Puntos adyacentes en el vector
                    dis=Algoritmo.calcularDistancia(v_puntos[i], v_puntos[j]);
                    if(dis < Algoritmo.dis_min) // Si la distancia entre esos 2 puntos es menor que la distancia minima
                    {
                        Algoritmo.dis_min=dis;
                        sol[0]=v_puntos[i];
                        sol[1]=v_puntos[j];
                    }
                }
            }
        }
    }
    return sol;
}
```

Pseudocódigo

```
funcion busquedaDyV(v_puntos: Punto[], izq: int, dch: int) return Punto[]
1     nPuntos=dch-izq+1;
2     mitad=izq+(dch-izq)/2;
3     i_izq=0;
4     i_dch=0;
5     si(nPuntos<4) entonces
6        sol=busquedaExhaustivaPoda(v_puntos, izq, dch);
7     sino
8        sol_izq=busquedaDyV(v_puntos, izq, mitad);
9        dis_izq=dis_min;
10       sol_dch=busquedaDyV(v_puntos, mitad+1, dch);
11       dis_dch=dis_min;
12       si(dis_izq<=dis_dch) entonces
13          dis_min=dis_izq;
14          sol=sol_izq;
15       sino
16          dis_min=dis_dch;
17          sol=sol_dch;
18       fsi
19       para(i_izq=mitad hasta izq con decremento -1)
20          dis=valor_absoluto(v_puntos[mitad].x - v_puntos[i_izq].x);
21          si(dis > dis_min) entonces
22             break;
23          fsi
24       fpara (i_izq)
25       para(i_dch=mitad+1 hasta dch con incremento 1)
26          dis=valor_absoluto(v_puntos[i_dch].x - v_puntos[mitad].x);
27          si(dis > dis_min) entonces
28             break;
29          fsi
30       fpara (i_dch)
31       para(i=mitad hasta i_izq+1 con decremento -1)
32          para(j=mitad+1 hasta i_dch-1 con incremento 1)
33             dis=valor_absoluto(v_puntos[j].x - v_puntos[i].x);
34             si(dis > dis_min) entonces
35                break;
36             sino
37                dis=calcularDistancia(v_puntos[i], v_puntos[j]);
38                si(dis < dis_min) entonces
39                   dis_min=dis;
40                   sol[0]=v_puntos[i];
41                   sol[1]=v_puntos[j];
42                fsi
43             fsi
44          fpara (j)
45       fpara (i)
46    fsi
47
48    return sol;
ffuncion
```

3.1. Caso Mejor

El caso mejor se da cuando todos los puntos del vector tienen una coordenada X distinta y la distancia mínima no está en la franja central, sino entre 2 puntos del subvector de la izquierda o entre 2 puntos del subvector de la derecha.
La ordenación previa del vector de puntos realizada por el algoritmo de ordenación **QuickSort** antes de llamar al algoritmo de búsqueda Divide y Vencerás, ordena los puntos por la coordenada X de menor a mayor, lo que permite dividir el vector de puntos recursivamente en dos subvectores ('sol_izq' y 'sol_dch') obteniendo así una mayor eficiencia, ya que los dos subvectores que obtenemos en cada división, son de menor talla que el vector original.
Este proceso se repite hasta que tenemos menos de 4 puntos en el vector y tiene un orden de complejidad de **O(n log n)**.
En este caso, en los dos bucles ***for*** (líneas 31, 32) se realiza una poda, ya que la distancia mínima está entre 2 puntos del subvector de la izquierda o entre 2 puntos del subvector de la derecha, y por tanto, 'dis' siempre es mayor que 'dis_min'. Al realizar esta poda, obtenemos un orden de complejidad próximo a **O(n)**.

Por tanto, podemos afirmar que para el caso mejor, el algoritmo de búsqueda Divide y Vencerás tiene un orden de complejidad de **O(n log n)**.

3.2. Caso Peor

El caso peor se da cuando todos los puntos del vector tienen la misma coordenada X, ya que el algoritmo de ordenación QuickSort no puede ordenar los puntos del vector por la coordenada X, y por tanto, el algoritmo dividirá el vector en dos subvectores, cuyos puntos no están ordenados por la coordenada X. Esto provocará que el algoritmo realice los dos últimos bucles ***for*** (líneas 31, 32) enteros siempre, debido a que todos los puntos están dentro de la franja central.
Como (v_puntos[i].x - mitad.x) siempre es 0, la distancia 'dis' siempre va a ser menor que la distancia mínima 'dis_min', y por tanto, no realizamos ninguna poda en el recorrido del subvector de la izquierda (línea 21), ni al recorrer el subvector de la derecha (línea 27), ni al recorrer los puntos de la franja central (línea 34).

Por tanto, podemos afirmar que para el caso peor el orden de complejidad es de **O(n^2)**.

3.3. Caso Medio

El orden de complejidad del caso medio depende del valor de la coordenada X de los puntos de los vectores de puntos generados aleatoriamente.
Si realizamos muchas podas, el orden de complejidad se acercará al caso mejor, y por lo tanto será próximo a **O(n log n)**.
No obstante, si prácticamente no realizamos ninguna poda, el orden de complejidad se acercará al caso peor, y por lo tanto será próximo a **O(n^2)**.

3.4. Gráficas

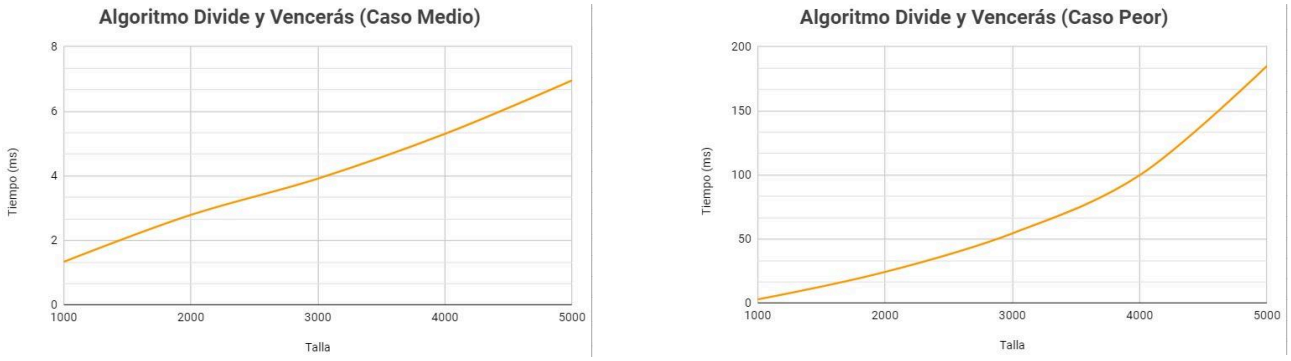


Imagen 4. Gráficas del Caso Medio y Caso Peor para el Algoritmo Divide y Vencerás.

4.- Algoritmo Divide y Vencerás Mejorado

A continuación, dejamos la solución **Divide y Vencerás Mejorado** en código JAVA y su pseudocódigo

JAVA

```java
 * @param v_puntos Es un vector de tipo 'Punto' que contiene todos los Puntos que han sido cargados desde el fichero
 * @param izq Es la posicion inicial (mas a la izquierda) del vector 'v_puntos'
 * @param dch Es la posicion final (mas a la derecha) del vector 'v_puntos'
 * @return Devuelve un vector 'sol' de tipo Punto con los 2 Puntos mas cercanos de todos los que hay cargados en el vector 'v_puntos'
 */
public static Punto[] busquedaDyV_mejorada(Punto v_puntos[], int izq, int dch)
{
    int nPuntos=dch-izq+1; // Nº de Puntos que hay en el Vector 'v_puntos'
    int mitad=izq+(dch-izq)/2; // Contiene la ultima posicion del Subvector de la Izquierda -> 0+(0+9)/2 = 4'5 = 4
    Punto sol_izq[]; // Vector solucion en el que cargaremos los 2 Puntos mas cercanos del Subvector de la izquierda
    Punto sol_dch[]; // Vector solucion en el que cargaremos los 2 Puntos mas cercanos del Subvector de la derecha
    Punto sol[]=new Punto[2]; // Vector solucion en el que cargaremos los 2 Puntos mas cercanos

    double dis_izq; // Distancia utilizada para comparar la distancia entre 2 Puntos del Subvector de la Izquierda en cada vuelta de bucle
    double dis_dch; // Distancia utilizada para comparar la distancia entre 2 Puntos del Subvector de la Derecha en cada vuelta de bucle
    double dis; // Distancia Actual calculada
    int i_izq=0; // Indice para recorrer el Subvector de la izquierda
    int i_dch=0; // Indice para recorrer el Subvector de la derecha

    if(nPuntos<4) // Si el vector 'v_puntos' tiene tantos Puntos como el Caso Base es mas eficiente utilizar el Algoritmo Exhaustivo con Poda
    {
        sol=Algoritmo.busquedaExhaustivaPoda(v_puntos, izq, dch);
    }
    else // Si el vector 'v_puntos' tiene mas Puntos que el Caso Base es mas eficiente utilizar la estrategia de Divide y Venceras
    {
        // Dividimos el Vector 'v_puntos' por la mitad en 2 Subvectores
        sol_izq=Algoritmo.busquedaDyV_mejorada(v_puntos, izq, mitad);
        dis_izq=Algoritmo.dis_min;

        sol_dch=Algoritmo.busquedaDyV_mejorada(v_puntos, mitad+1, dch);
        dis_dch=Algoritmo.dis_min;

        if(dis_izq<=dis_dch) // Si la distancia que hay entre los 2 Puntos del Subvector Izquierdo es menor que la distancia que hay entre los 2 Puntos del Subvector Derecho
        {
            Algoritmo.dis_min=dis_izq;
            sol=sol_izq;
        }
        else // Si la distancia que hay entre los 2 Puntos del Subvector Derecho es menor que la distancia que hay entre los 2 Puntos del Subvector Izquierdo
        {
            Algoritmo.dis_min=dis_dch;
            sol=sol_dch;
        }
        // Calculamos los Puntos mas cercanos a la frontera, es decir, a la mitad del Vector Original
        // Calculamos la distancia de los Puntos que quedan en el Subvector Izquierdo
        for(i_izq=mitad; i_izq>=izq; i_izq--)
        {
            // Como el Vector inicial esta ordenado, si llegamos a un valor de 'i' en el que la distancia es mayor que la minima, los siguientes valores del Subvector tambien tendran una distancia mayor que la minima
            dis=Math.abs(v_puntos[mitad].getX()-v_puntos[i_izq].getX());
            if(dis > Algoritmo.dis_min) break;
        }

        // Calculamos la distancia de los Puntos que quedan en el Subvector Derecho
        for(i_dch=mitad+1; i_dch<=dch; i_dch++)
        {
            // Como el Vector inicial esta ordenado, si llegamos a un valor de 'i' en el que la distancia es mayor que la minima, los siguientes valores del Subvector tambien tendran una distancia mayor que la minima
            dis=Math.abs(v_puntos[i_dch].getX()-v_puntos[mitad].getX());
            if(dis > Algoritmo.dis_min) break;
        }

        // Tratamos la franja Central, es decir, vamos recorriendo el vector de puntos desde la posicion podada 'i_izq' hasta 'i_dch' y cargamos los Puntos en un Vector 'v_franja'
        ArrayList<Punto> v_franja=new ArrayList<>();
        for(int i=i_izq+1; i<i_dch; i++)
        {
            v_franja.add(v_puntos[i]);
        }

        // Convertimos el ArrayList<Punto> en Punto[] 'v_central' para poder pasarselo como parametro al QuickSort()
        Punto[] v_central=v_franja.toArray(new Punto[v_franja.size()]);

        // Ordenamos el Vector 'v_centtral' por el eje 'Y'
        Algoritmo.QuickSort(v_central, 0, v_central.length-1, 'y');

        // Como la distancia minima entre 2 puntos puede estar entre un Punto de la Franja Izquierda y un Punto de la Franja Derecha
        // Recorremos el vector 'v_puntos' hasta las posiciones podadas y comparamos la distancia de los Puntos del Subvector de la Izquierda (i) y del Subvector de la Derecha (j)
        // En el caso de que la distancia minima se repita, la distancia minima sera la del primer Punto del Vector, si NO queremos que sea asi -> if(dis >= dis_min)
        for(int i=0; i<v_central.length; i++) // Recorremos el Subvector de la Izquierda
        {
            for(int j=i+1; j<v_central.length && (Math.abs(v_central[j].getY()-v_central[i].getY()) < Algoritmo.dis_min); j++) // Recorremos el Subvector de la Derecha
            {
                // Comparamos la distancia de los 2 Puntos adyacentes en el vector
                dis = Algoritmo.calcularDistancia(v_central[i], v_central[j]);
                if (dis < Algoritmo.dis_min) // Si la distancia entre esos 2 puntos es menor que la distancia minima
                {
                    Algoritmo.dis_min = dis;
                    sol[0] = v_central[i];
                    sol[1] = v_central[j];
                }
            }
        }
    }
    return sol;
}
```

Pseudocódigo

```
funcion busquedaDyV_mejorada(v_puntos: Punto[], izq: int, dch: int) return Punto[]
1     nPuntos=dch-izq+1;
2     mitad=izq+(dch-izq)/2;
3     i_izq=0;
4     i_dch=0;
5     v_franja: ArrayList<Punto>;
6     v_central: Punto[];
7     si(nPuntos<4) entonces
8        sol=busquedaExhaustivaPoda(v_puntos, izq, dch);
9     sino
10       sol_izq=busquedaDyV_mejorada(v_puntos, izq, mitad);
11       dis_izq=dis_min;
12       sol_dch=busquedaDyV_mejorada(v_puntos, mitad+1, dch);
13       dis_dch=dis_min;
14       si(dis_izq<=dis_dch) entonces
15          dis_min=dis_izq;
16          sol=sol_izq;
17       sino
18          dis_min=dis_dch;
19          sol=sol_dch;
20       para(i_izq=mitad hasta =izq con decremento 1)
21          dis=valor_absoluto(v_puntos[mitad].x - v_puntos[i_izq].x);
22          si(dis > dis_min) entonces
23             break;
24          fsi
25       fpara (i_izq)
26       para(i_dch=mitad+1 hasta dch con incremento 1)
27          dis=valor_absoluto(v_puntos[i_dch].x - v_puntos[mitad].x);
28          si(dis > dis_min) entonces
29             break;
30          fsi
31       fpara(i_dch)
32       para(i=i_izq+1 hasta i_dch-1 con incremento 1)
33          v_franja.add(v_puntos[i]);
34       fpara (i)
35       v_central=v_franja.toArray(Punto[v_franja.size()]);
36       QuickSort(v_central, 0, v_central.length-1, 'y');
37       para(i=0 hasta v_central.length-1 con incremento 1)
38          para(j=i+1 hasta (v_central.length-1) AND
                              (valor_absoluto(v_central[j].y - v_central[i].y) < dis_min)) con incremento 1)
39             dis = calcularDistancia(v_central[i], v_central[j]);
40             si(dis < dis_min) entonces
41                dis_min = dis;
42                sol[0] = v_central[i];
43                sol[1] = v_central[j];
44             fsi
45          fpara(j)
46       fpara (i)
47    fsi
48    return sol;
ffuncion
```

4.1. Caso Mejor

El caso mejor se da en el mismo caso que el algoritmo de búsqueda divide y vencerás, esto es, cuando todos los puntos del vector tienen una coordenada X distinta y la distancia mínima no está en la franja central, sino entre 2 puntos del subvector de la izquierda o entre 2 puntos del subvector de la derecha.

Es muy parecido al del algoritmo divide y vencerás, la principal diferencia está en que antes de recorrer los puntos que están en la franja central 'v_central', ordenamos los puntos de dicho vector por la coordenada Y de menor a mayor mediante el algoritmo de ordenación QuickSort, lo que permite que aunque la coordenada X sea la misma, los puntos estén ordenados en el vector y siempre podamos realizar alguna poda para hacer menos comparaciones, y por tanto, obtener así una mayor eficiencia.

Podemos afirmar que el orden de complejidad del algoritmo de búsqueda divide y vencerás mejorado es de **O(n log n)**, sin embargo, comparándolo con el algoritmo de búsqueda divide y vencerás, su tiempo de ejecución será mayor, ya que debe ordenar recursivamente los puntos del vector de la franja central 'v_central' por la coordenada Y.

4.2. Caso Peor

El caso peor se da en el mismo caso que el algoritmo de búsqueda divide y vencerás, es decir, cuando todos los puntos del vector tienen la misma coordenada X.
Sin embargo, en el algoritmo de búsqueda divide y vencerás mejorado, la principal diferencia está en que antes de recorrer los puntos que están en la franja central 'v_central', ordenamos los puntos de dicho vector por la coordenada Y de menor a mayor mediante el algoritmo de ordenación QuickSort, lo que permite que aunque la coordenada X sea la misma, los puntos estén ordenados en el vector.
Esto permite que en el último bucle ***for*** (línea 38) podamos realizar alguna poda siempre que v_central[j].y - v_central[i].y sea menor que 'dis_min', permitiéndonos hacer menos comparaciones, y por tanto, obtener así una mayor eficiencia.

Podemos afirmar que en el caso peor el algoritmo de búsqueda divide y vencerás mejorado tiene un orden de complejidad de **O(n log n)** y un tiempo de ejecución mucho menor que todos los algoritmos de búsqueda explicados anteriormente.

4.3. Caso Medio

El orden de complejidad del caso medio es el mismo que el del caso mejor y que el del caso peor, es decir, **O(n log n)**.
El tiempo de ejecución dependerá de la distribución uniforme de los puntos del vector respecto a la coordenada X, ya que cuanto mayor sea el número de puntos que tienen la misma coordenada X, menor será el tiempo de ejecución del algoritmo.

4.4. Gráficas

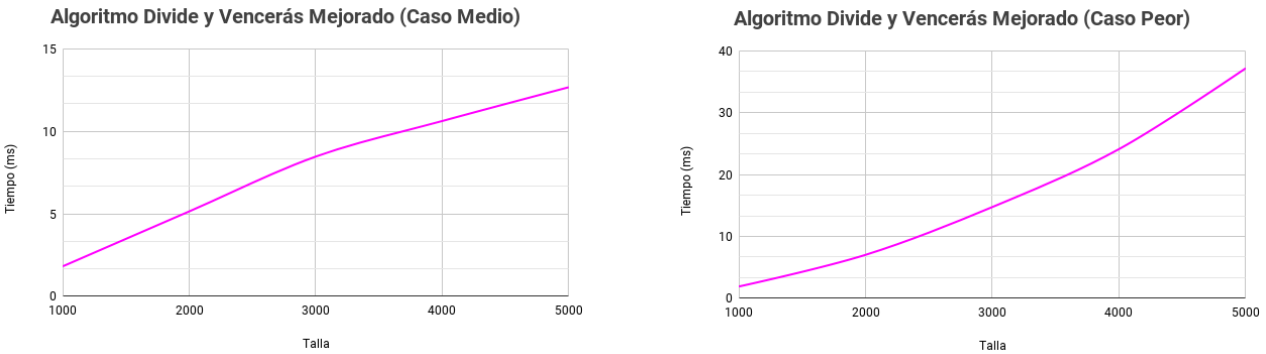
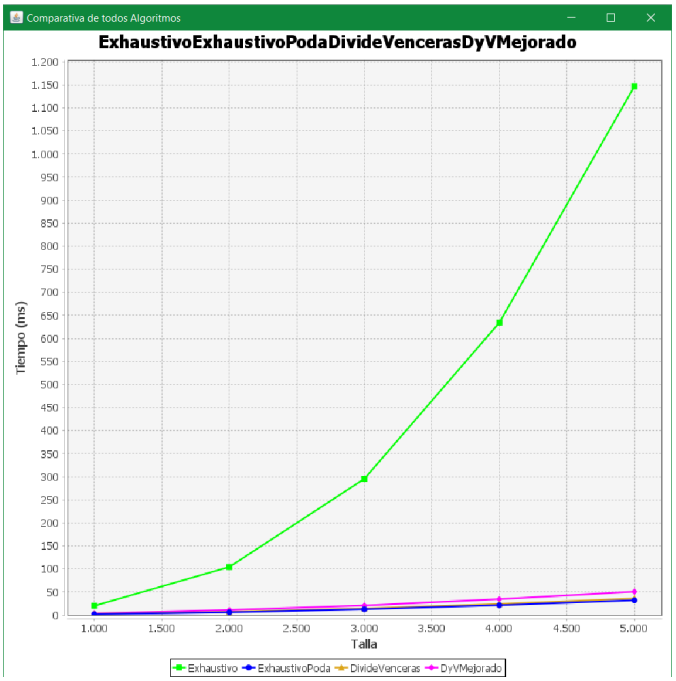


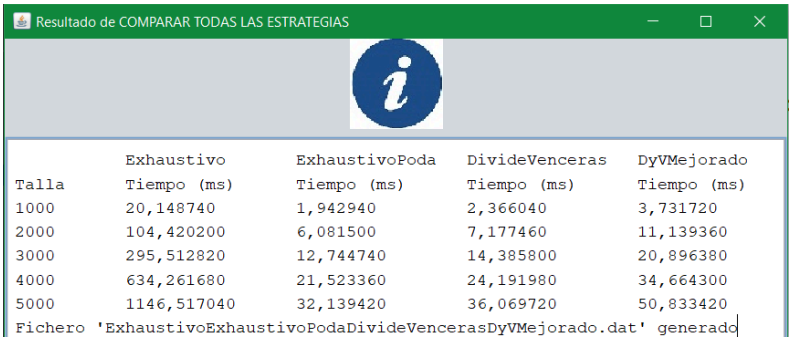
Imagen 5. Gráficas del Caso Medio y Caso Peor para el Algoritmo Divide y Vencerás Mejorado.

4.- Comparativa de todas las Estrategias

Análisis de Coste Teórico				
Caso	**Exhaustivo**	**Exhaustivo con Poda**	**Divide y Vencerás**	**Divide y Vencerás Mejorado**
Mejor	$O(n^2)$	O(n log n)	O(n log n)	O(n log n)
Medio	$O(n^2)$	O(n log n) / $O(n^2)$	O(n log n) / $O(n^2)$	O(n log n)
Peor	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(n log n)

4.1. Caso Medio





Talla	Exhaustivo Tiempo (ms)	ExhaustivoPoda Tiempo (ms)	DivideVenceras Tiempo (ms)	DyVMejorado Tiempo (ms)
1000	20,148740	1,942940	2,366040	3,731720
2000	104,420200	6,081500	7,177460	11,139360
3000	295,512820	12,744740	14,385800	20,896380
4000	634,261680	21,523360	24,191980	34,664300
5000	1146,517040	32,139420	36,069720	50,833420

Fichero 'ExhaustivoExhaustivoPodaDivideVencerasDyVMejorado.dat' generado

Para el caso medio, podemos observar que el orden de eficiencia de los algoritmos de menor a mayor coste en tiempo es: ExhaustivoPoda < DivideVencerás < DyVMejorado < Exhaustivo

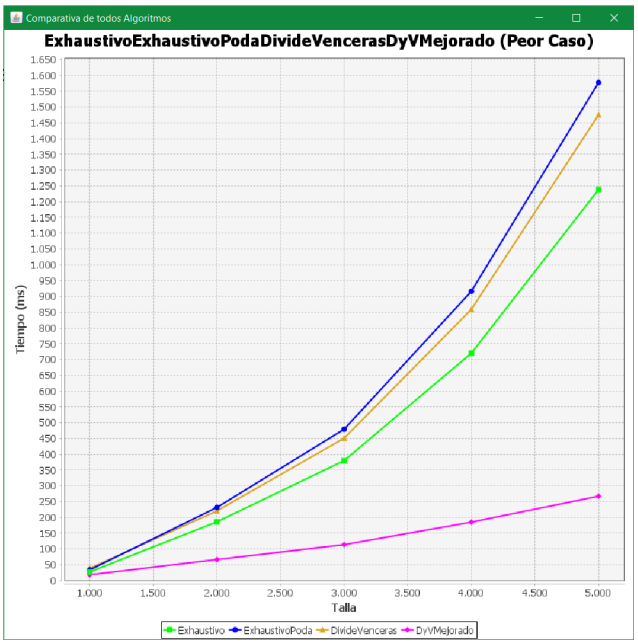
En vectores de talla 1000, el tiempo de ejecución de los algoritmos *ExhaustivoPoda*, *DivideVencerás*, y *DyVMejorado* es muy similar, ya que los 3 están por debajo de los 4ms (0’ 004s). Sin embargo, el algoritmo *Exhaustivo*, al realizar más comparaciones se vuelve más lento, obteniendo la solución en 20ms (0’ 020s).

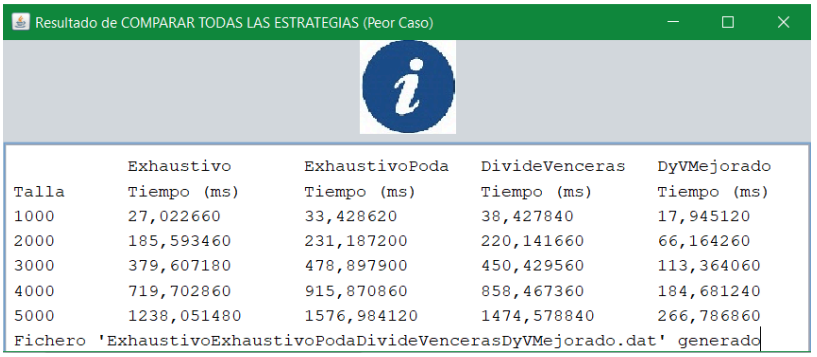
Si bien es cierto, que es un tiempo muy bajo, comparando el algoritmo *Exhaustivo* con los otros 3, llega a ser entre 5 veces más lento que los algoritmos ExhaustivoPoda y DivideVencerás y 10 veces más lento que el algoritmo DyVMejorado en vectores de talla pequeña.

En vectores de talla 5000, el tiempo de ejecución de los algoritmos *ExhaustivoPoda* y *DivideVencerás* es muy similar, ya que ambos están en torno a los 35ms (0’ 035s). El algoritmo *DyVMejorado* se vuelve más lento al tener que ordenar el vector de puntos por el eje Y recursivamente, obteniendo la solución en unos 50ms (0’ 050s).
Por otro lado, el algoritmo *Exhaustivo*, al tener un orden de complejidad de **$O(n^2)$**, cuanto mayor sea la talla del vector, su coste temporal aumentará de forma cuadrática, obteniendo la solución en unos 1146ms (1’ 146s).

Comparando el algoritmo *ExhaustivoPoda* con el resto de algoritmos, es muy similar al algoritmo DivideVencerás, casi el doble de rápido que el algoritmo *DyVMejorado* y 573 veces más rápido que el algoritmo *Exhaustivo*.

4.2. Caso Peor





Talla	Exhaustivo Tiempo (ms)	ExhaustivoPoda Tiempo (ms)	DivideVenceras Tiempo (ms)	DyVMejorado Tiempo (ms)
1000	27,022660	33,428620	38,427840	17,945120
2000	185,593460	231,187200	220,141660	66,164260
3000	379,607180	478,897900	450,429560	113,364060
4000	719,702860	915,870860	858,467360	184,681240
5000	1238,051480	1576,984120	1474,578840	266,786860

Fichero 'ExhaustivoExhaustivoPodaDivideVencerasDyVMejorado.dat' generado

Para el caso medio, podemos observar que el orden de eficiencia de los algoritmos de menor a mayor coste en tiempo es: DyVMejorado < Exhaustivo < DivideVencerás < ExhaustivoPoda

En vectores de talla 1000, el tiempo de ejecución de los algoritmos *ExhaustivoPoda* y *DivideVencerás* es similar, ya que ambos están en torno a los 35ms (0’ 035s). En el algoritmo *DivideVencerás*, como los puntos no están ordenados por la coordenada X, debe recorrer siempre el vector entero y no realiza ninguna poda en los subvectores, por lo que se vuelve el más lento de todos, obteniendo la solución en unos 38ms (0’ 038s). Sorprendentemente, el algoritmo Exhaustivo es el segundo mejor de todos, obteniendo la solución en unos 27ms (0’ 027s).
En el algoritmo *DyVMejorado* se ordenan los puntos por el la coordenada Y recursivamente, lo que permite realizar podas para reducir el número de comparaciones y vueltas de bucle, obteniendo la solución en unos 18ms (0’ 018s).

Comparando el algoritmo *DyVMejorado* con el resto de algoritmos, es casi 2 veces más rápido que los algoritmos *ExhaustivoPoda DivideVencerás* y no llega a ser el doble de rápido que el algoritmo *Exhaustivo*.

En vectores de talla 5000, el tiempo de ejecución del algoritmo *DyVMejorado* es el mejor de todos, obteniendo la solución en tan solo 266ms (0’ 266s). Por otro lado, los algoritmos *DivideVencerás* y *ExhaustivoPoda* son muy similares, ya que ambos están en torno a los 1500ms (1’ 500s). Sorprendentemente, el algoritmo Exhaustivo es el segundo mejor de todos, obteniendo la solución en 1238ms (1’ 238s).

Comparando el algoritmo *DyVMejorado* con el resto de algoritmos, es casi 5 veces más rápido que el algoritmo *Exhaustivo* y casi 6 veces más rápido que los algoritmos *DivideVencerás* y *ExhaustivoPoda*.

5.- Datos técnicos del equipo

El equipo en el que se han realizado los cálculos y del que hemos obtenido los datos temporales de la eficiencia de cada algoritmo es distinto al equipo en el que se realizará la defensa de la práctica (los resultados podrían variar por las especificaciones del equipo).

Los componentes del ordenador son los siguientes:

- **CPU:** i7-6700 (3’4 GHz up to 4GHz) con 4 núcleos y 8 Hilos
- **GPU:** NVIDIA GTX 1660 Ti 6GB GDDR6
- **RAM:** 32GB RAM DDR4

6.- Conclusión y valoración personal de la práctica

Con la realización de esta práctica hemos aprendido a visualizar la eficiencia de los diferentes algoritmos de búsqueda. Además, hemos adquirido la capacidad de crear gráficas para representar la eficiencia de los algoritmos en JFreeChart y a programar código JAVA en NetBeans, IDE que nunca antes habíamos utilizado. Esto ha sido bastante útil, ya que este tipo de prácticas facilita la comprensión de los algoritmos vistos en las clases de teoría. El hecho de realizar la práctica en JAVA ha reforzado mis conocimientos de este lenguaje de programación, que aprovecharé para otras asignaturas.