Práctica 2: Algoritmos Divide y Vencerás

Alejandro Trujillo Caballero

7 de enero de 2015

Índice

1. Introducción

En esta prática se pretende estudiar la eficiencia temporal de un algoritmo divide y vencerás frente a la solución exahustiva para el mismo problema. Para ello se solucionará el problema de encontrar los dos puntos más cercanos en una nube de puntos.

La codificación exahustiva recorrerá cada pareja posible de puntos y calcular1á la distancia, almacenando la menor encontrada.

La codificación divide y vencerás dividirá la nube en cuatro cuadrantes a partir de un punto pivote y volverá a ejecutarse sobre cada cuadrante de manera recursiva hasta encontrar un caso base.

2. Análisis teórico

2.1. Algoritmo exahustivo

```
1  public double exhaustiva (Punto[] puntos) {
2      double distancia;
3      double distanciaMinima = Double.PositiveInfinity;
4      for (int i = 0; i < puntos.Length; i++) {
5          for (int j = i + 1; j < puntos.Length; j++) {
6              distancia = puntos [i].distancia (puntos [j]);
7              if (distanciaMinima > distancia) {
8                  distanciaMinima = distancia;
9              }
10         }
11     }
12     return distanciaMinima;
13 }
```

Calculamos el tiempo de ejecución del algoritmo:

$$T(n) = A - \sum_{i=1}^{n} \sum_{j=i}^{n} B$$

Siendo A las operaciones que se realizan antes y después de los bucles for (operaciones de orden constante), y B las que se realizan en el interior del segundo for (también constantes). Desarrollando la expresión:

$$T(n) = A - B \sum_{i=1}^{n} (n-i) = A + B \left(\sum_{=1}^{n} n - \sum_{=1}^{n} i \right) = A + B \left(n^2 - \frac{n(1+n)}{2} \right) = A + B \left(\frac{n^2}{2} - \frac{n}{2} \right)$$

En el caso mejor sólo entraría en el if una vez y en el caso peor entraría siempre, por lo que para ambos casos A y B son constantes y obtenemos que el algoritmo es de orden cuadrático:

$$T(n) \in O(n^2)$$

2.2. Algoritmo divide y vencerás

```
public double DyV_cuatroCuadrantes (Punto[] puntos, double distanciaMin) {

    double distanciaMinima = distanciaMin;
    double distancia;
    Punto pivote;

    List<Punto>[] cuadrantes = new List<Punto>[4];
    for (int i = 0; i < 4; i++) {
        cuadrantes [i] = new List<Punto> ();
    }


    if (puntos.Length == 2) {
        distanciaMinima = puntos [0].distancia (puntos [1]);
    } else {

        pivote = partition (puntos, ref cuadrantes);

        foreach (List<Punto> cuadrante in cuadrantes) {
            if (cuadrante.Count > 1) {
                distancia = DyV_cuatroCuadrantes (cuadrante.ToArray (),
                    distanciaMinima);
                if (distanciaMinima > distancia)
                    distanciaMinima = distancia;
            }
        }

    //fronteraX
        List<Punto> regionX = new List<Punto> ();

        foreach (List<Punto> cuadrante in cuadrantes) {
            foreach (Punto punto in cuadrante) {
                if (Math.Abs (punto.Y - pivote.Y) < distanciaMinima) {
                    regionX.Add (punto);
                }
            }
        }

        if (regionX.Count > 1 && regionX.Count < puntos.Length) {
            distancia = DyV_cuatroCuadrantes (regionX.ToArray (),
                distanciaMinima);
            if (distancia < distanciaMinima) {
                distanciaMinima = distancia;
            }
        }

    //fronteraY
        List<Punto> regionY = new List<Punto> ();

        foreach (List<Punto> cuadrante in cuadrantes) {
            foreach (Punto punto in cuadrante) {
                if (Math.Abs (punto.X - pivote.X) < distanciaMinima) {
```

```
51                          regionY.Add (punto);
52                      }
53                  }
54              }
55
56              if (regionY.Count > 1 && regionY.Count < puntos.Length) {
57                  distancia = DyV_cuatroCuadrantes (regionY.ToArray (),
                        distanciaMinima);
58                  if (distancia < distanciaMinima) {
59                      distanciaMinima = distancia;
60                  }
61              }
62
63          }
64          return distanciaMinima;
65      }
```

El coste temporal de este algoritmo viene dado por la expresión:

$$T(n) = \begin{cases} B & si \quad n \leq 2 \\ \\ T_{partition} + 4T(n/4) + T_{fronteras} + A & si \quad n > 2 \end{cases}$$

Donde:

- A = Operaciones elementales generales (declaraciones, creación de variables...). Orden constante.
- B = Coste del caso base, en nuestro caso el cálculo de la distancia entre dos puntos. Orden constante.
- $4T(n/4)$ = Coste de las cuatro llamadas recursivas de tamaño $n/4$ (cuadrantes).
- $T_{partition}$ = Coste de la llamada a partition.
- $T_{fronteras}$ = Coste del cálculo de fronteras entre los cuadrantes.

La función partition no está especificada en esta memoria ya que su funcionamiento consiste simplemente en tomar el primer punto como pivote y recorrer la nube, asignando cada punto al cuadrante correspondiente en base al pivote. Tenemos por tanto $T_{partition} = n$.

En caso de que el pivote elegido no divida los puntos, la función elige uno diferente hasta que sí se produce división, sin embargo no vamos a considerarlo a la hora de realizar los cálculos teóricos ya que el orden de la función seguiría siendo lineal y solo serviría para dificultar el desarrollo matemático.

En el mejor caso, el algoritmo encontrará las distancia mínima al realizar las llamadas recursivas y esta será suficientemente pequeña como para que no se realice ninguna llamada recursiva durante el cálculo de fronteras obteniendo $T_{fronteras} = 2n$ derivado de la identificación de las mismas. Tenemos que:

$$T(n) = \begin{cases} B & si \quad n \leq 2 \\ \\ n + 4T(n/4) + 2n + A & si \quad n > 2 \end{cases}$$

Si aplicamos el teorema maestro[1] para el cálculo de órdenes de algoritmos divide y vencerás:

[1] Presente en el tema 1 de teoría de la asignatura.

$$a = 4, k = 1, b = 4 \Rightarrow T(n) \in O(n \log n)$$

En el caso peor, podríamos encontrar una nube de puntos distribuida de tal forma (por ejemplo cuatro puntos formando un cuadrado) que el algoritmo entre en un bucle sin salida ya que el algoritmo partition devolvería un punto en cada cuadrante, no se obtendría distancia mínima válida de la primera parte del algoritmo y al identificar las fronteras los cuatro puntos serían considerados frontera y volvería a realizarse exactamente la misma llamada al algoritmo, entrando en un bucle infinito.

Para evitar esto se ha introducido un caso base diferente en el algoritmo, consistente en que para nubes menores de 60 puntos se realiza eelcálculo exahustivo de la distancia. ¿Por qué es esto una mejora? En primer lugar, para valores pequeños de n, el algoritmo exahustivo es aproximadamente igual de rápido que el divide y vencerás pero enormemente más ligero en memoria y en segundo lugar, la pila de llamadas recursivas es limitada y para nubes muy grandes este caso base con cálculo exahustivo reduce enormemente la carga en esa pila, permitiendo trabajar con nubes mucho mayores sin obtener desbordamientos.

Por otra parte, modificamos el caso peor que anteriormente nos llevaba a un bucle infinito para que ahora se ejecute también en tiempo logarítmico.

Podemos hacer una aproximación del tiempo medio de ejecución del algoritmo suponiendo que en cada llamada se consideran fronteras un cuarto de los puntos, obteniendo:

$$T(n) = \begin{cases} n^2 & si \quad n \leq 60 \\ \\ n + 4T(n/4) + 2n + 2T(n/4) + A = 3n + 6T(n/4) & si \quad n > 60 \end{cases}$$

Aunque n^2 no es una costante, podemos aplicar el teorema maestro ya que sabemos que en nuestro caso está acotada superiormente por 60^2 que sí lo es y obtenemos:

$$a = 6, k = 1, b = 4 \Rightarrow a > b^k \Rightarrow T(n) \in O(n^{\log_4 6}) = O(n^{1,2925})$$

3. Resultados experimentales

3.1. Algoritmo exahustivo

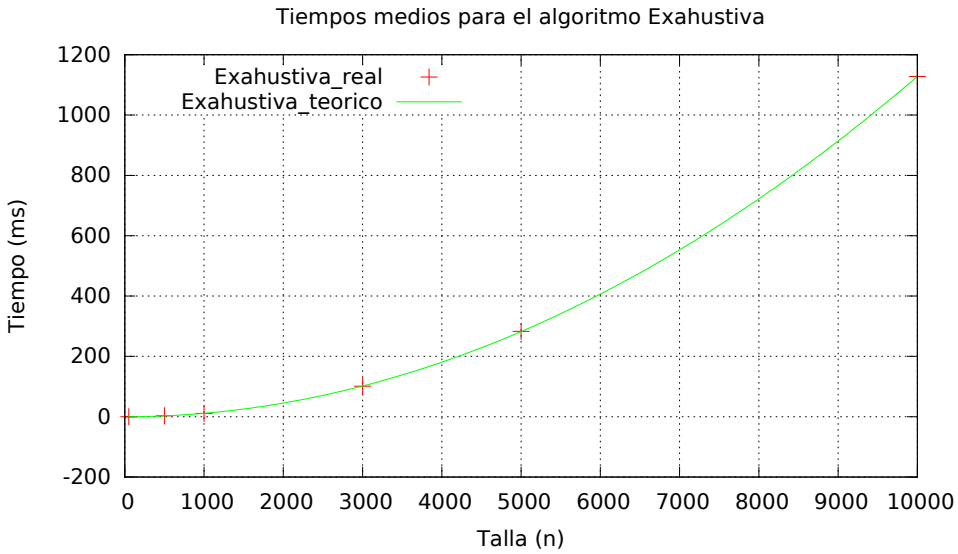

Figura 1: Ejecuciones del algoritmo exahustivo para diferentes tamaños de la nube

Podemos ver en la figura 1 como las ejecuciones realizadas se ajustan perfectamente a la curva cuadrática teórica.

3.2. Algoritmo divide y vencerás

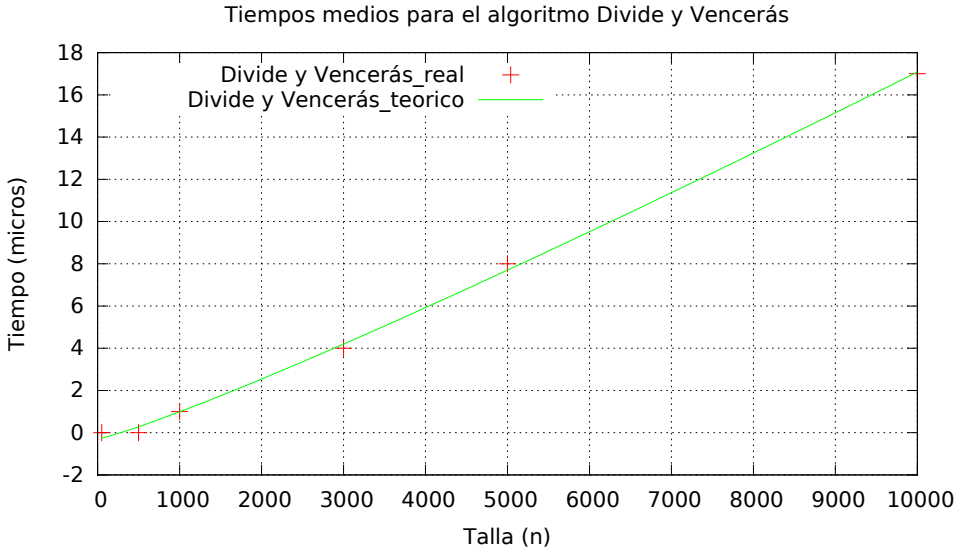

Figura 2: Ejecuciones del algoritmo divide y vencerás para diferentes tamaños de la nube

Podemos ver en la figura 2 que la ejecución del algoritmo divide y vencerás se ajusta a una curva de orden $O(n \log n)$.

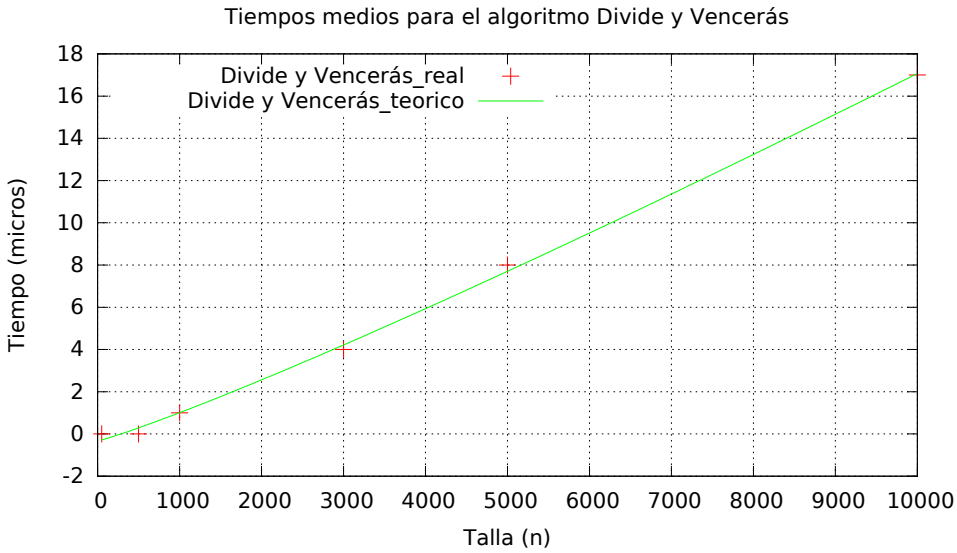

Figura 3: Ajuste de la ejecución divide y vencerás al caso medio aproximado

Podemos ver también en la figura 3 que se ajusta de forma razonable a la curva aproximada calculada anteriormente en el análisis teórico.

3.3. Comparación de algoritmos

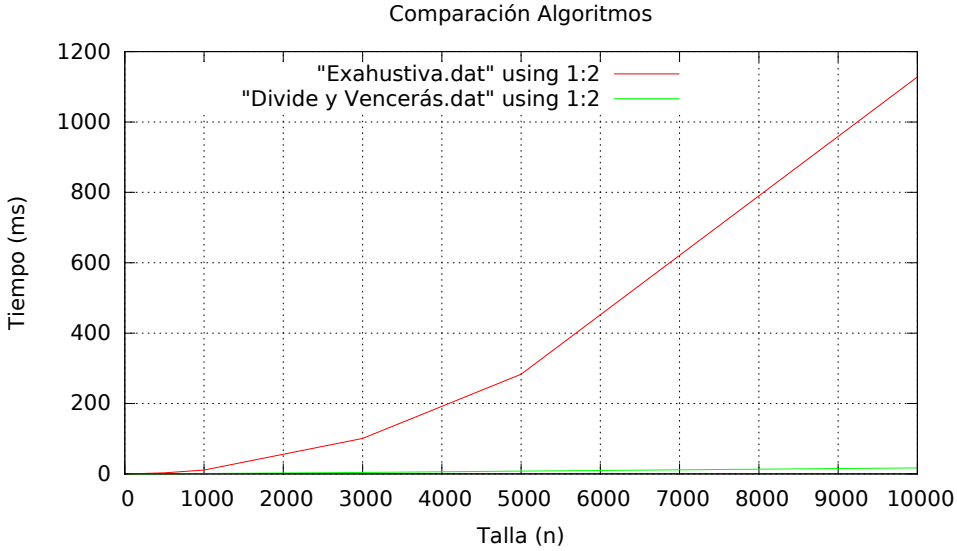

Figura 4: Comparación de algoritmos exahustivo y divide y vencerás

En la figura 4 se puede confirmar de forma evidente el objetivo de la práctica, al dibujar en la misma gráfica ambas ejecuciones se aprecia como el algoritmo divide y venceras es tremendamente más rápido que el exahustivo, y sobre todo, como aumenta la diferencia al aumentar el tamaño de los datos.

Para evidenciar aún más la diferencia se han buscado para qué tamaños de la nube el algoritmo tarda entre 2 y 3 minutos:

- Para 110000 puntos, el agoritmo exahustivo tarda 2,3 minutos, el divide y vencerás 0,2 segundos.
- Para 25 millones, el algoritmo divide y vencerás tarda 2,5 minutos, el exahustivo no se ha ejecutado para esa cantidad de puntos.
- Para 50 millones, el algoritmo divide y vencerás tarda 5,9 minutos.

4. Conclusiones y mejoras

El algoritmo divide y vencerás es notoriamente más rapido que el exahustivo tal y como se esperaba, sin embargo necesita mucha más memoria tanto para la pila como para datos y arrays auxiliares en cada llamada.

Como posibles mejoras, estaría bien encontrar alguna forma de controlar el problema de los bucles infinitos sin recurrir a exahustivo o calcular el valor óptimo del valor de caso base (el actual de 60 es algo arbitrario).