Universidad de Valladolid

Algoritmo Divide y vencerás

Par de puntos más cercano en espacio n-dimensional

Sergio García Prado
October 18, 2015

1 Introducción

El problema que se va a analizar se basa en encontrar los dos puntos más cercanos entre un conjunto de puntos pertenecientes en un espacio n-dimensional. Es una condición obligatoria que todos los puntos pertenezca a la misma dimensión ya que de no ser así no tendría sentido comparar sus distancias.

Existen distintos enfoques para resolver este problema. El mas simple pero a la vez menos eficiente de todos se basa en comparar todos los puntos con todos e ir guardando los dos que menor distancia tienen entre sí. Esta solución tiene un coste de $O(n2)$ algo que podemos mejorar si nos damos cuenta de algunos detalles mediante algoritmos del tipo divide y vencerás.

Motivos por los que usar divide y vencerás

1. Los puntos mas cercanos en el espacio por la propia definición de cercanía van a estar en una región próxima del espacio. Este es el motivo por el cual nos podemos ahorrar comparar dos puntos que están muy alejados en el mapa.

2. Si encontramos un mínimo en un subconjunto del espacio y este lo es también para todos los subconjuntos que contienen a este, entonces habremos encontrado el mínimo de todo el espacio.

2 Solución

La solución que se ha escogido es la de realizar particiones binarias en el espacio recursivamente hasta tener subconjuntos de pequeño tamaño (En la implementación propuesta como ejemplo se ha fijado en conjuntos de 10 puntos.) para después compararlos utilizando un algoritmo boraz que nos asegura el mínimo local de cada subconjunto. Lo siguiente es quedarse con el mínimo de las dos particiones binarias y analizar los puntos que se quedaron en la frontera de las dos particiones ya que puede darse el caso de que el par de puntos con distancia mínima contuviese el punto 1 en la particion 1 y el punto 2 en la partición 2. De no ser por la parte de combinación estos casos no se estudiarían y podría darse el caso de que el mínimo encontrado no fuera el real.

Las explicaciones se van a exponer en un espacio de 3 dimensiones pero estas son extrapolables cualquier número de dimensiones.

2.1 Divide

Lo que intentamos conseguir al dividir el espacio en particiones binarias recursivamente es agrupar los puntos que están más próximos para así intentar prescindir del mayor número de comparaciones innecesarias. Para ello se pueden tomar distintos enfoques:

- El primero de ellos consiste en hacer la partición siempre en la misma dimensión. Este enfoque tiene la ventaja de que tan solo hay que ordenar los elementos una vez, ya que al no tener que cambiar de dimensión la ordenación se mantiene. Pero aún así esto es menos eficiente ya que no se consigue la meta deseada que era agrupar los puntos cercanos en conjuntos. En

este caso al solo depender de una de las dimensiones en las otras pueden tener valores muy diferentes por lo que al producirse la fusión se tienen que evaluar muchos más puntos. Esto se ilustra en la figura 1.

- La segunda solución consiste en que en cada nivel de recursión se cambie la dimensión en la que se particionan los puntos, lo que conlleva a una reordenación de los mismos respecto a dicha dimensión. La carga de trabajo en este caso es mayor pero la división que se consigue es mucho más homogénea en cuanto a distancia lo que nos da una gran ventaja al fusionar los distintos subconjuntos. Esto se ilustra en la figura 2.

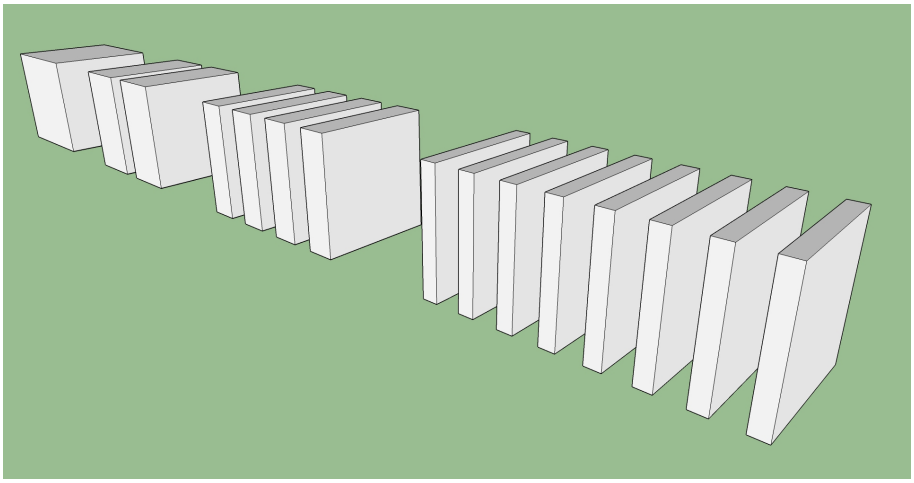

Figure 1: Particionamiento en la misma dimensión

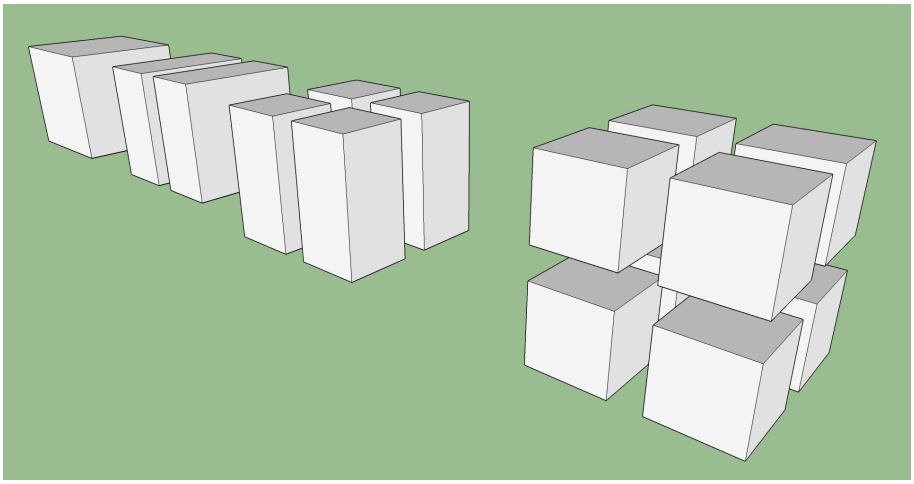

Figure 2: Particionamiento en distintas dimensiones

2.2 Venceras

3 Pseudocódigo

4 Análisis de crecimiento

Algorithm 1 Euclid's algorithm

```
1: procedure CLOSESTPAIR(PLIST, i, dim)                    ▷ The g.c.d. of a and b
2:
3:     if len(PLIST) > 0 then
4:
5:         i ← (i + 1) mod dim
6:
7:         PLIST ← sort(i, PLIST)
8:
9:         leftPLIST ← PLIST[: len(PLIST)/2]
10:
11:        rightPLIST ← PLIST[len(PLIST)/2 :]
12:
13:        leftCPAIR ← CLOSESTPAIR(leftPLIST, i, dim)
14:
15:        rightCPAIR ← CLOSESTPAIR(rightPLIST, i, dim)
16:
17:        CPAIR ← min(leftCPAIR, righCPAIR)
18:
19:        distance ← CPAIR.distance()
20:
21:        borderPLIST ← closePLIST(distance, leftPLIST, rightPLIST)
22:
23:        borderCPAIR ← CLOSESTPAIR(borderPLIST, i, dim)
24:
25:        CPAIR ← min(CPAIR, borderCPAIR)
26:
27:    else
28:
29:        CPAIR ← borazPair(PLIST)
30:
31:    end if
32:
33:    return CPAIR                                        ▷ The gcd is b
34:
35: end procedure
```