АНАЛИЗ НА РЕШЕНИЕТО НА ЗАДАЧА CLOSEST POINTS

Задачата Closest Points изискваше в абсолютно прав вид стандартна, но не особено популярна в състезателната информатика структура данни: KD-дърво. Нещо повече, задачата беше допълнително опростена от факта, че точките са генерирани по случаен начин – това позволява да не прилагаме различни оптимизации за специфични случаи и пак да си гарантираме бързи (O(logN)) заявки.

Състезателите, които нямаха идея как да подходят, можеха да хванат 20 лесни точки чрез сортиране и двоично търсене. 30 точки пък бяха предвидени за решение, което разглежда само 100-те най-близки точки по всяка координата за всяко куери.
За да се хванат 50 точки трябваше да се знае алгоритъма за намиране на най-близка точка в 2Д пространство. Алгоритъмът е базиран на техниката "разделяй-и-владей" и всеки път разделя пространството по X или по Y, като игнорира голям брой от точките (около половината, при равномерно-разпределени точки). Как става игнорирането? Ако например търсим най-близката точка до тази с координати (42, 13), ако досегашният ни най-добър кандидат е на разстояние 8 и правим разделяне по X в 30, то сме сигурни, че никои от точките с по-малък X не може да подобри резултата, тъй като само по X разстоянието е поне 12.

Интересното е, че алгоритъмът за 100 точки не е много по-различен. Реално строи двоично дърво за търсене, във всеки връх на което стои една от дадените N точки. В "лявото" поддърво са всички точки с по-малки координати по някое от измеренията, докато в дясната – останалите. Така реално комбинираме горния алгоритъм с двоично дърво за търсене. Както казахме, при равномерно-разпределени точки на всяка стъпка бихме могли да разделяме точките на приблизително половина (поравно във всяко поддърво) ако изберем медианата за разделител. Разбира се, тъй като имаме K измерения трябва за всеки връх от дървото да решим по кое от тях ще разделяме точките. Един прост вариант е да вземаме височината на текущия връх по модул K и да избираме съответната координата. (Има и по-сложни начини на избор, които дават малко по-добър резултат, но те не бяха нужни за задачата.)

Самото строене на дървото става по много подобен начин, по който строим двоично дърво за търсене. В зависимост от имплементацията можем да постигнем различни сложности, но може би една от най-простите е със задоволителните O(N*logN). Друг вариант е да се възползваме, че координатите са относително малки и да сортираме точките по всяка от размерностите с Counting Sort, постигайки сложност O(K*N).
Всяка от заявките пък се спуска надолу по дървото (което е с дълбочина O(logN), при равномерно-разпределени точки). Забележете, че както и при алгоритъма за 2D точки може да се наложи да слезем *и в двата* сина на някои от върховете. При произволно-разпределени точки, обаче, очакваното търсене намира достатъчно бързо близък кандидат, така че да спре търсенето нататък и да остане с очаквана сложност O(logN).
Друг вариант, с който можеха да се хванат доста на брой точки (70-80, а и дори 100 точки в зависимост от имплементацията) беше с разпределяне на точките в "бъкети". Самото търсене се извършва като се намери в кой (K-мерен) бъкет попада query-то и търсене само в него (и някои от съседните), което отново значително намалява пространството на търсене. Подобно решение може да се реализира и с Quad Tree.

Автор: Александър Георгиев