3ª Aula Prática – Divisão e Conquista

Instruções

- Faça download do ficheiro **cal_fp03_CLion.zip** da página da disciplina e descomprima-o (contém a pasta **lib,** a pasta **Tests** com os ficheiros **tests.cpp**, **Point.h**, **Point.cpp**, **NearestPoints.h**, **NearestPoints.cpp**, e os ficheiros **CMakeLists** e **main.cpp**, assim como ficheiros texto com informação de input para os algoritmos)
- No CLion, abra um ***projeto***, selecionando a pasta que contém os ficheiros do ponto anterior.
- Efetuar "*Load CMake Project*" sobre o ficheiro *CMakeLists.txt*
- Execute o projeto (**Run**)
- Note que os ***testes unitários deste projeto podem estar comentados***. Se for este o caso, retire os comentários à medida que vai implementando os testes.
- ***Deverá realizar esta ficha respeitando a ordem das alíneas***.
- Efetue a implementação no ficheiro nos respetivos ficheiros ***.cpp***.
- Nota importante: para ler ficheiros de texto em modo I/O, deverá configurar a sua localização no CLion, redefinindo a variável do ambiente IDE "Working Directory", a parir do menu Run > Edit Configurations… > Working Direcory.

Enunciado

1. Cálculo dos pontos mais próximos

Suponha que *P* é uma lista de pontos num plano. Se *p1=(x1,y1)* e *p2=(x2,y2),* a distância euclidiana entre *p1* e *p2* é dada por:

$$[(x_1 - x_2)^2 + (y_1 - y_2)^2]^{1/2}$$

São fornecidos alguns ficheiros de dados com um número (potência de 2) de pontos aleatórios. A ideia é encontrar os pontos mais próximos. No caso de existirem dois pontos na mesma posição esses serão os mais próximos, com a distância zero.

a. A função ***Result nearestPoints_BF(vector<Ponto> &vp)***
retorna um objeto com os pontos mais próximos pertencentes ao vetor *vp* e a distância entre eles. Implemente esta função com um algoritmo de força bruta, corra os testes e o tempo necessário para as pesquisas. Verifique se o tempo é aproximadamente quadrático $O(N^2)$ no tamanho dos dados.

b. A função ***Result nearestPoints_DC(vector<Ponto> &v)***
tem um objetivo semelhante à anterior, mas seguindo um algoritmo de divisão e conquista como o descrito mais abaixo (mas sem implementar a parte final que obriga a ter duas listas). Implemente

a função, execute os testes e compare os tempos de execução com os da alínea *a*. Verifique se o tempo é aproximadamente $O(N \log^2 N)$ no tamanho dos dados.

c. A função ***Result nearestPoints_DC_MT(vector<Ponto> &v)*** tem um objetivo semelhante à anterior, mas tira partido de processamento paralelo (*multi threading)*, à semelhança do exemplo de MergeSort abordado nas aulas teóricas. O número de *threads* a usar é definido pela função *setNumThreads*. Implementa o suporte para *multi threading*. Corra os testes e verifique qual é o número de *threads* ótimo.

d. Indique um invariante e variante de ciclo principal da função da alínea *a)*, e mostre que cumprem as propriedades que permitem provar que o algoritmo está correto.

e. Demonstre que a complexidade temporal do algoritmo de divisão e conquista da alínea *b)* é $O(N \log^2 N)$.

f. Implemente a função ***Result nearestPoints_BF_SortByX(vector<Ponto> &vp)*** que refina o algoritmo de força bruta com uma ordenação prévia dos pontos segundo o eixo dos X. Verifique que o tempo execução é muito bom quando os pontos estão dispostos de forma mais ou menos aleatória, mas não quando os pontos diferem apenas na coordenada X.

Algoritmo Divisão e Conquista para calcular os pontos mais próximos

(traduzido de M.A. Weiss, "Data Structures and Algorithms Analysis in C++", 3rd edition –cap. 10, pág. 430-435)

Suponha que P é uma lista de pontos num plano. Se $p_1=(x_1,y_1)$ e $p_2=(x_2,y_2)$, a distância euclidiana entre *p1* e *p2* é dada por:

$$[(x_1 - x_2)^2 + (y_1 - y_2)^2]^{1/2}$$

A ideia é encontrar os pontos mais próximos. No caso de existirem dois pontos na mesma posição esses serão os mais próximos, com a distância zero.

Se existem *N* pontos, então existem *N(N-1)/2* pares de distâncias. Podemos verificá-las todas, com um algoritmo muito simples de pesquisa exaustiva (ou força bruta), mas com complexidade temporal $O(N^2)$. Com um algoritmo tipo divisão e conquista, como o descrito seguidamente, consegue-se garantir uma complexidade temporal de *O(N log N)*.

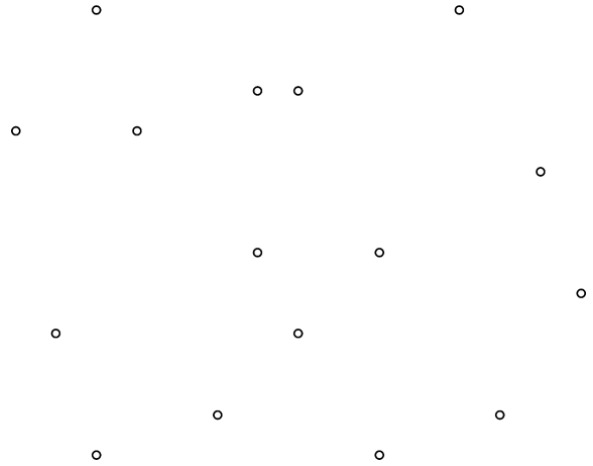

Figura 1 - Um pequeno conjunto *P* de pontos

A figura 1 mostra um pequeno conjunto *P* de pontos. Se os pontos estiverem ordenados pela abcissa (*x*) podemos desenhar uma linha imaginária vertical que divide o conjunto em duas metades, P_L e P_R. Com esta divisão, os dois pontos mais próximos estarão ambos em P_L ou ambos em P_R ou um em P_L e outro em P_R. Podemos chamar a estas distâncias d_L, d_R e d_C, tal como se mostra na figura 2.

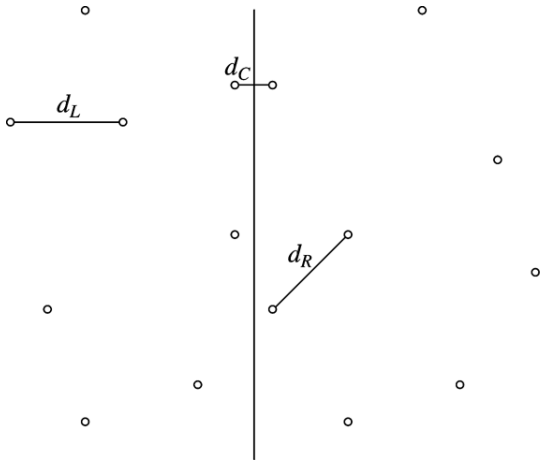

Figura 2 - Conjunto *P* dividido em P_L e P_R com as distâncias mínimas indicadas

O cálculo de d_L e d_R pode ser recursivo. O problema desloca-se, então, para o cálculo de d_C. Para garantir um algoritmo de complexidade *O(N log N)*, o necessário para ordenar os valores, deve ser possível calcular d_C em *O(N)*.

Considerando $\delta = \min(d_L, d_R)$, observamos que só é necessário calcular d_C se diminuir δ. Sendo assim, os dois pontos que definem d_C devem estar a menos de δ da linha divisória; designaremos esta área como **faixa**. Como mostra a figura 3, esta consideração limita o número de pontos que devem ser analisados – no caso da figura $\delta = d_R$.

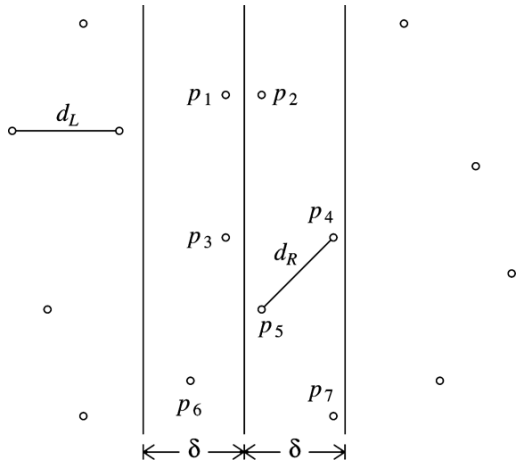

Figura 3 - Duas pistas contendo todos os pontos considerados para a faixa d_C

Existem duas estratégias para calcular d_C. Para conjuntos muito alargados de pontos uniformemente distribuídos, o número de pontos esperado contidos dentro desta faixa é muito reduzido – em média estarão $O(\sqrt{N})$. Então, nesta área podemos usar o algoritmo de força bruta em todos os pontos, em tempo $O(N)$. O pseudo código para esta estratégia é apresentado na figura 4.

```
// Points are all in the strip

for( i = 0; i < numPointsInStrip; i++ )
    for( j = i + 1; j < numPointsInStrip; j++ )
        if( dist(pi, pj) < δ )
            δ = dist(pi, pj);
```

Figura 4 - Algoritmo de força bruta para calcular *min(δ, dC)*

No pior caso, todos os pontos podem estar na faixa, pelo que esta estratégia não funcionará em tempo linear. É necessário olhar com atenção para o problema para melhorar o algoritmo: as coordenadas *y* de dois pontos que definem d_C devem diferir, no máximo, δ. De outra forma $d_C > \delta$. Supondo que os pontos da faixa estão ordenados por *y*, se as coordenadas *y* de *pi* e p_j divergirem mais do que δ, então passaremos para o ponto p_{i+1}. Esta modificação simples está implementada no algoritmo da figura 5.

```
// Points are all in the strip and sorted by y-coordinate

for( i = 0; i < numPointsInStrip; i++ )
    for( j = i + 1; j < numPointsInStrip; j++ )
        if( pi and pj's y-coordinates differ by more than δ )
            break;          // Go to next pi.
        else
        if( dist(pi, pj) < δ )
            δ = dist(pi, pj);
```

Figura 5 - Cálculo *refinado* de *min(δ, d_C)*

Este simples teste extra tem um efeito significativo na evolução do algoritmo, porque para cada ponto p_i poucos pontos p_j são examinados (se as suas coordenadas divergirem mais do que δ o ciclo *for* interno é terminado. A figura 6 mostra, por exemplo, que para o ponto *p3*, apenas os pontos *p4* e *p5* dentro da faixa ficam a menos da distância vertical δ.

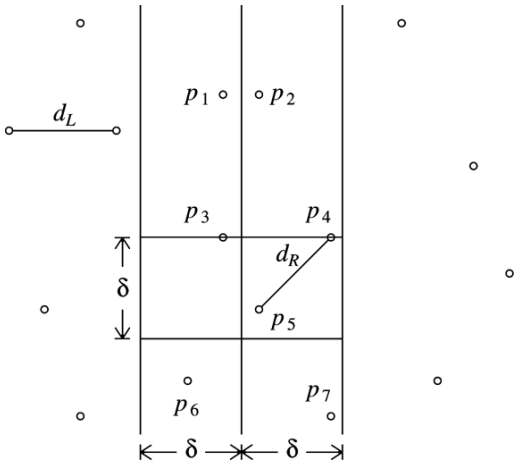

Figura 6 - Apenas os pontos p_4 e p_5 são considerados no segundo ciclo *for*

No pior caso, para qualquer ponto p_i serão considerados sete pontos p_j. A razão é simples de entender: estes pontos devem estar contidos num quadrado de d por d na metade esquerda da faixa ou num quadrado δ por δ na metade direita da faixa. Por outro lado todos os pontos em cada quadrado δ por δ estão separados, no mínimo, por δ. No pior caso, cada quadrado contém quatro pontos, um em cada vértice. Um desses pontos é p_i, deixando no máximo sete pontos para serem considerados. A figura 7 tenta ilustrar este caso. Mesmo que os pontos p_{L2} e p_{R1} tenham as mesmas coordenadas podem ser pontos diferentes. Nesta análise, o que é importante é que o número de pontos no rectângulo λ por 2λ é *O(1)*.

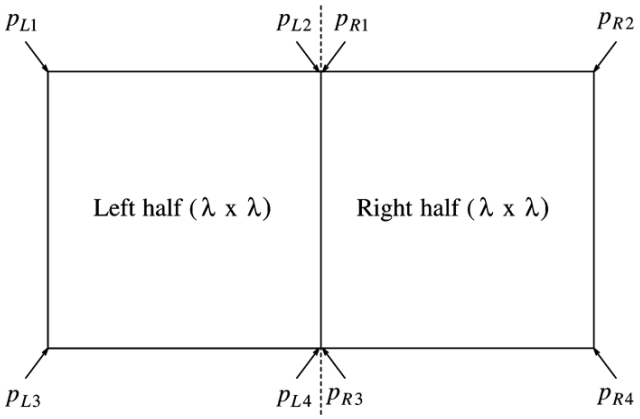

Figura 7 - No máximo oito pontos ficam no rectângulo, partilhando cada ponto duas coordenadas

Devido ao facto de, no máximo, sete pontos serem considerados para cada p_i, o tempo para calcular um d_C melhor que δ é *O(N)*. Assim, parece que obtemos uma solução *O(N log N)* para os pontos mais próximos, baseada na chamada recursiva de duas metades mais o tempo linear para combinar os dois resultados.

Contudo ainda não obtivemos uma solução efetiva *O(N log N)*. Assumimos que a lista de pontos está ordenada. Se realizarmos esta ordenação em cada chamada recursiva, então temos um trabalho extra de *O(N log N)*, o que dá um algoritmo $O(N \log^2 N)$. Isto não é mau de todo se compararmos com a complexidade $O(N^2)$ do algoritmo de força bruta.

No entanto, é relativamente fácil reduzir a complexidade de cada chamada recursiva para *O(N)*, garantindo um algoritmo de complexidade *O(N log N)*. A ideia é manter duas listas. Uma é a lista de pontos ordenados pela coordenada *x*, e a outra a lista de pontos ordenados pela coordenada *y*. Isto pressupõe um primeiro passo de ordenação com um custo temporal de *O(N log N)*. Se referenciarmos estas listas por *P* e *Q*,

respectivamente, P_L e Q_L são as listas passadas para a chamada recursiva da metade esquerda e P_R e Q_R as listas passadas para a chamada recursiva da metade direita. A lista P é facilmente separada a meio. Uma vez conhecida a linha divisória, percorremos a lista Q sequencialmente colocando cada elemento em Q_L ou Q_R como apropriado. Facilmente se verifica que as listas Q_L e Q_R estão ordenadas por coordenada y. Quando a chamada recursiva retorna, percorremos a lista Q e retiramos todos os pontos cujas coordenadas x não fiquem dentro da faixa. Então a lista Q apenas contém os pontos que ficam dentro da faixa, já devidamente ordenados pela coordenada y.

Esta estratégia garante todo o algoritmo com complexidade *O(NlogN)*, porque apenas é realizado processamento extra com complexidade *O(N)*.