ALGORITMOS PARA CÁLCULO DA MENOR DISTÂNCIA EUCLIDIANA EM UM CONJUNTO DE PONTOS NO ESPAÇO BIDIMENSIONAL

Felipe Fontenele de Ávila Magalhães, Mateus Vitor Pereira Lana, Thiago Oliveira de Santana

Curso de Bacharelado em Ciência da Computação – Universidade Federal de Ouro Preto (UFOP) – Campus Morro do Cruzeiro
35400-000 – Ouro Preto – MG – Brasil

felipe.fontenele@decom.ufop.br

mateus_lana7@hotmail.com

thiago63.guitarrista@gmail.com

Abstract. *This paper describes a study of the problem of the shortest distance Euclidean in a two-dimensional space. Two algorithms were implemented to solve the problem, using the brute force technique in one and the paradigm of division and conquest in the other. Afterwards, both the complexity and the empirical analysis of the algorithms were performed.*

Resumo. *Este artigo descreve um estudo à respeito do problema da menor distância euclidiana em um espaço bidimensional. Foram implementados dois algoritmos para a resolução do problema, utilizando a técnica da força bruta em um e o paradigma de divisão e conquista no outro. Posteriormente foi realizada uma análise tanto de complexidade quanto empírica dos algoritmos.*

Palavras Chave: Distância Euclidiana, Força Bruta, Divisão e Conquista

1. Introdução

Um dos conceitos básicos da Geometria é que a menor distância entre dois pontos é dada por uma reta. Na Geometria Analítica, esses pontos recebem coordenadas no plano cartesiano e, por meio dessas coordenadas, podemos encontrar o valor da distância euclidiana. O cálculo da distância euclidiana, apesar de se tratar de algo bem simples, pode ser aplicado em diversificados contextos, como nos serviços de GPS por exemplo.

Neste trabalho, buscamos encontrar a menor distância euclidiana em um conjunto de pontos finitos em um espaço bidimensional. Os pontos são gerados aleatoriamente e salvos em um arquivo texto que posteriormente é lido pelo programa e o conjunto dos pontos é salvo em um "vector" do tipo "Ponto", que é uma struct que possui as coordenadas. Como já foi dito, codificamos duas implementações para a resolução do problema, com divisão e conquista e força bruta. Utilizamos a linguagem C++ na implementação, por questão de preferência.

O objetivo do trabalho é compreender como funcionam os dois algoritmos implementados e compará-los em relação a complexidade e tempo de execução. Para isso, fizemos a análise de complexidade e a análise empírica das implementações. Além disso os algoritmos têm inserido neles uma função para medir os tempos de execução de cada um.

A organização do trabalho foi dividida nos seguintes tópicos: fazer a criação dos números aleatórios e salvar em um arquivo .txt, fazer a leitura do arquivo, implementação do algoritmo Força Bruta, e por último, mas não menos importante, a implementação do algoritmo Divisão e Conquista.

Portanto, observamos que apesar do algoritmo Força Bruta possuir uma complexidade pior que a Divisão e Conquista, o mesmo apresenta uma execução mais rápida para os pontos gerados pelo grupo.

2. Algoritmo de Força Bruta

```
1  double forcaBruta(vector<Ponto>& pontos, int tam, Ponto* ponto1, Ponto* ponto2){
2      double menorDist, dist;
3      double x1, y1, x2, y2;
4      int i, j, cont=0;
5      menorDist = DBL_MAX;
6      for(i=0; i<tam-1; i++){
7          x1 = pontos[i].x;
8          y1 = pontos[i].y;
9          for(j=i+1; j<tam; j++){
10             x2 = pontos[j].x;
11             y2 = pontos[j].y;
12             if(x2 - x1 > menorDist){
13                 break;
14             }
15             dist = distancia(x1,y1,x2,y2);
16             if(dist < menorDist){
17                 menorDist = dist;
18                 *ponto1 = pontos[i];
19                 *ponto2 = pontos[j];
20             }
21         }
22     }
23     return menorDist;
24 }
```

2.1 Descrição do algoritmo

A função ***forcaBruta*** recebe como parâmetro o vetor de pontos com seu respectivo tamanho e duas variáveis do tipo ***TPonto*** (***ponto1, ponto2***), que são passadas por referência, para guardarem os pontos que possuem a menor distância entre eles.

Dentro da função ***forcaBruta*** declaramos as variáveis necessárias e inicializamos a variável ***menorDist*** (que representará a menor distância entre os pontos a cada iteração) com um valor suficientemente grande. Em seguida temos o algoritmo

propriamente dito, que basicamente trata-se de dois comandos ***for*** alinhados, o contador mais externo varia de 0 até a quantidade de pontos subtraída de uma unidade (até o penúltimo ponto); já o contador interno inicia-se com valor atual do contador externo acrescido de uma unidade e varia até a quantidade de pontos (até o último ponto).

Dentro do ***for*** interno é realizado o cálculo da distância que é dado pela fórmula:

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

e o resultado é guardado na variável ***dist***. Em seguida é feita a comparação entre a distância que foi calculada na iteração atual (***dist***) e a menor distância encontrada até o momento (***menorDist***). Caso a distância encontrada na iteração atual seja menor, o valor de ***menorDist*** é atualizado e os pontos envolvidos nela são guardados em ***ponto1*** e ***ponto2***.

Devido ao pré-processamento realizado no vetor de pontos, que simplesmente ordena o mesmo de acordo com a coordenada x, foi adicionada uma melhoria no algoritmo de força bruta que diminui consideravelmente o número de vezes que o cálculo da distância é realizado. O primeiro comando ***if*** dentro do ***for*** interno é o responsável pela melhoria, o que ele faz é verificar se a diferença entre as coordenadas x (**x2** - **x1**) dos dois pontos que estão sendo analisados a cada iteração é menor que a maior distância obtida até o momento (**menorDist**). Caso isso seja satisfeito, o ***for*** interno é quebrado e o cálculo da distância não é realizado.

Ao final da execução o valor da menor distância armazenado em ***menorDist*** é retornado e o algoritmo de força bruta se encerra.

2.2 Análise de complexidade

Primeiramente, no pré-processamento é feito a ordenação das coordenadas em tempo **O(n*logn)**.Em relação as criações de variáveis, comparações e atribuições considera-se a complexidade como **O(1)**. O ***for*** externo (linha 6) é executado **n-1** vezes, considerando **n** como a quantidade de pontos. Sendo assim, ao desconsiderar a constante, a complexidade no ***for*** externo fica **O(n)**. O ***for*** interno (linha 9) será executado a mesma quantidade de vezes que o externo (n-1), sendo assim desconsiderando as constantes teremos **O(n)** no ***for*** interno, assim como no externo. Portanto, devido ao fato de termos os comandos ***for*** alinhados a complexidade do algoritmo será $\mathbf{O(n^2)}$.

2.3 Avaliação experimental

Primeiramente, todos os testes realizados foram feitos em uma máquina Intel I5, 500GB de armazenamento, utilizando o sistema operacional Linux do laboratório Com 22, para caso desejar averiguar a veracidade dos dados.

A estrutura dos testes foi a seguinte: as dez quantidades de pontos distintas variam de mil a dez mil, aumentado em mil unidades a cada teste, ou seja, o primeiro teste foi realizado com mil pontos, o segundo com dois mil pontos, o terceiro com três mil pontos e assim sucessivamente até chegar em dez mil pontos. Os intervalos para as coordenadas dos pontos iniciam espaçados em dez mil unidades, no intervalo de -5000 a

5000, e aumentam o intervalo em dez mil unidades até o intervalo ser de cem mil unidades, mantendo a simetria entre números positivos e negativos.

Para cada quantidade de pontos associada a um intervalo específico foi guardado o tempo de execução do algoritmo em segundos. Em seguida foi calculada a média dos tempos de execução do algoritmo para cada quantidade de pontos. Os resultados obtidos nesses experimentos, com o algoritmo de força bruta pode ser visto nas tabelas abaixo:

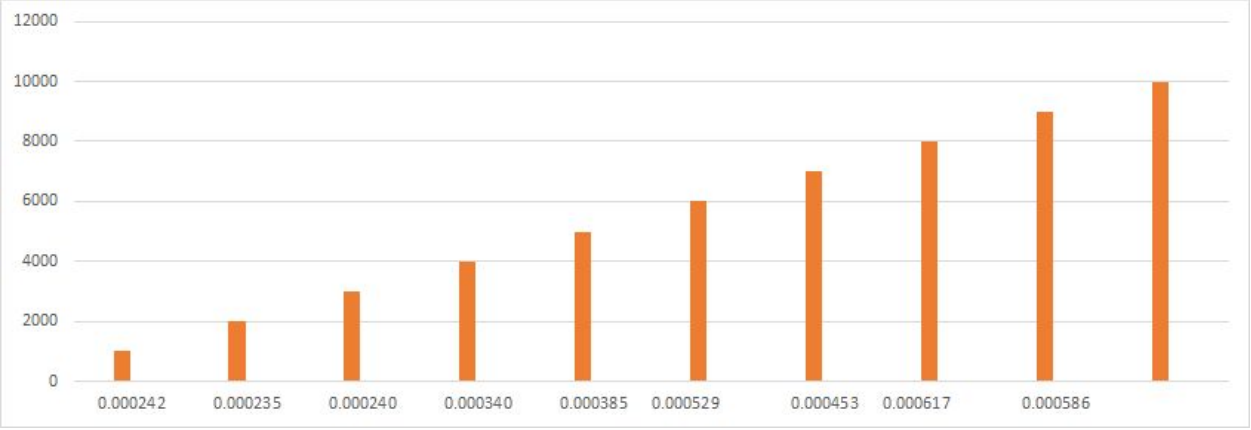
Tabela 1: As linhas representam as quantidades de pontos e as colunas representam, em unidade de milhar, os intervalos dos pontos gerados.

	-5 a 5	**-10 a 10**	**-15 a 15**	**-20 a 20**	**-25 a 25**	**-30 a 30**	**-35 a 35**	**-40 a 40**	**-45 a 45**	**-50 a 50**
1000	0.00014	0.00011	0.00009	0.00037	0.00012	0.00037	0.00089	0.00018	0.00009	0.00006
2000	0.00035	0.00023	0.00033	0.00025	0.00021	0.00010	0.00026	0.00028	0.00014	0.00020
3000	0.00025	0.00034	0.00027	0.00012	0.00037	0.00017	0.00014	0.00023	0.00026	0.00025
4000	0.00034	0.00046	0.00031	0.00039	0.00046	0.00033	0.00033	0.00041	0.00013	0.00024
5000	0.00049	0.00031	0.00046	0.00037	0.00041	0.00042	0.00043	0.00031	0.00029	0.00036
6000	0.00064	0.00048	0.00056	0.00036	0.00070	0.00074	0.00042	0.00047	0.00037	0.00055
7000	0.00036	0.00046	0.00039	0.00064	0.00041	0.00018	0.00054	0.00042	0.00061	0.00052
8000	0.00047	0.00071	0.00084	0.00076	0.00054	0.00065	0.00065	0.00064	0.00038	0.00053
9000	0.00083	0.00062	0.00107	0.00048	0.00054	0.00052	0.00088	0.00071	0.00061	0.00056
10000	0.00086	0.00078	0.00058	0.00059	0.00058	0.00081	0.00071	0.00053	0.00078	0.00055

Tabela 2: Média e intervalo de confiança para cada quantidade de pontos.

Quantidade de Pontos	Média (Tempo)	Intervalo de Confiança
1000	0.000242	0.000085 - 0.000399
2000	0.000235	0.000187 - 0.000283
3000	0.000240	0.000190 - 0.000290
4000	0.000340	0.000278 - 0.000402
5000	0.000385	0.000343 – 0.000427
6000	0.000529	0.000447 – 0.000611
7000	0.000453	0.000370 – 0.000536
8000	0.000617	0.000531 – 0,000703
9000	0.000586	0.000453 – 0.000718
10000	0.000677	0.000600 – 0.000754

Grafico 1: Pontos por média de tempo.



Após analisar o gráfico é possível perceber que o tempo aumenta de acordo com a quantidade de pontos.

3. Algoritmo de Divisão e Conquista

```
double divisaoConquista(Ponto *pontoX, Ponto *pontoY,int tam, Ponto* ponto1, Ponto* ponto2){
    if(tam <= 3)
        return forcaBruta(pontoX, tam, ponto1, ponto2);

    int meio = tam/2;
    Ponto pontoMeio = pontoX[meio];
    Ponto pontoYEsq[meio+1];
    Ponto pontoYDir[tam-meio-1];
    int li=0, ri=0;
    for(int i = 0; i < tam; i++){
        if(pontoY[i].x <= pontoMeio.x)
            pontoYEsq[li++] = pontoY[i];
        else
            pontoYDir[ri++] = pontoY[i];
    }
    double distEsq = divisaoConquista(pontoX,pontoYEsq,meio,ponto1,ponto2);
    double distDir = divisaoConquista(pontoX+meio,pontoYDir,tam-meio,ponto1,ponto2);
    double d;
    if(distEsq < distDir){
        d = distEsq;
        *ponto1 = pontoX[meio];
        *ponto2 = pontoY[meio];
    }else{
        d = distDir;
        *ponto1 = pontoX[tam-meio];
        *ponto2 = pontoY[tam-meio];
    }
    Ponto faixa[tam];
    int j = 0;
    for(int i = 0; i < tam; i++){
        if(abs(pontoY[i].x - pontoMeio.x) < d){
            faixa[j] = pontoY[i];
            j++;
        }
    }
    return min(d, menorFaixa(faixa,j,d,ponto1,ponto2));
}
```

```
double menorFaixa(Ponto *faixa, int tam, double d, Ponto *ponto1, Ponto *ponto2){
    double menor = d;

    for(int i = 0; i < tam; ++i){
        for(int j = i+1; j < tam && (faixa[j].y - faixa[i].y) < menor; ++j){
            if(distancia(faixa[i].x,faixa[i].y,faixa[j].x,faixa[j].y) < menor){
                menor = distancia(faixa[i].x,faixa[i].y,faixa[j].x,faixa[j].y);
                *ponto1 = faixa[i];
                *ponto2 = faixa[j];
            }
        }
    }
    return menor;
}
```

3.1. Descrição do algoritmo

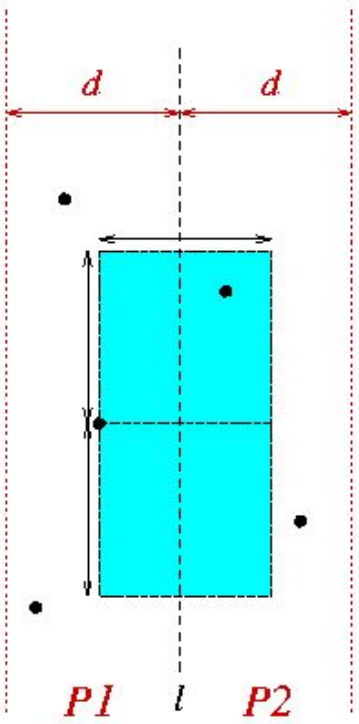
A função ***divisaoConquista*** recebe como parâmetro dois vetores de pontos (**pontoX**,**pontoY**) com seus respectivos tamanhos e duas variáveis do tipo ***TPonto*** (***ponto1, ponto2***), que são passadas por referência, para guardarem os pontos que possuem a menor distância entre eles. No pré-processamento que é ocorrido no

“Main.cpp”,o vetor **pontoX** é ordenado pelo x e o vetor **pontoY** é ordenado pelo y, devido a este ser um dos passos iniciais para a resolução do problema em questão.

O algoritmo segue os seguintes passos:

1. Dividir todos os pontos em duas metades.
2. Encontrar recursivamente as menores distâncias em ambas metades (esquerda e direita).
3. Salvar a menor distância na variável **d**.
4. Criar um vetor **faixa** que armazene todos os pontos que estão no máximo a uma distância **d** da linha intermediária dividindo os dois conjuntos.
5. Encontrar a menor distância da **faixa**.
6. Retornar a menor distância entre a distância **d** e a menor da **faixa**.

Seguindo a abordagem acima, um aspecto interessante é que se o vetor **faixa** estiver ordenado de acordo com a coordenada y, podemos encontrar a menor distância nessa faixa em O(n). Uma ideia central do algoritmo é predefinir todos os pontos de acordo com as coordenadas y. Quando fazemos chamadas recursivas, precisamos dividir os pontos do vetor **pontoY** de acordo com a linha vertical(imagem abaixo). Podemos fazer isso simplesmente processando cada ponto e comparando sua coordenada x com a coordenada x da linha média.



3.2 Análise de complexidade

Em relação as criações de variáveis, comparações e atribuições considera-se a complexidade como **O(1)**. Assumindo a complexidade do algoritmo Divisão e Conquista em **T(n)**, inicialmente o mesmo divide todos os pontos em dois conjuntos e chamadas recursivas para os mesmos. Depois de dividir, encontra a **faixa** em tempo **O(n)** conforme explicado anteriormente. Além disso, o algoritmo leva **O(n)** para dividir o conjunto das coordenadas do vetor **pontoY** em torno da linha média. Finalmente, encontra os pontos mais próximos na faixa de **O(n)**. Portanto, **T(n)** pode ser expresso da seguinte forma:

$T(n) = 2T(n/2) + O(n) + O(n) + O(n)$, que é igual a: $T(n) = 2T(n/2) + O(n)$

Efetuando o cálculo da complexidade, seja por expansão da fórmula ou por meio do teorema Mestre, obtemos que a complexidade do algoritmo é: **O(n*log(n))**

3.3 Avaliação experimental

Primeiramente, todos os testes realizados foram feitos em uma máquina Intel I5, 500GB de armazenamento, utilizando o sistema operacional Linux do laboratório Com 22, para caso desejar averiguar a veracidade dos dados.

A estrutura dos testes foi a seguinte: as dez quantidades de pontos distintas variam de mil a dez mil, aumentado em mil unidades a cada teste, ou seja, o primeiro teste foi realizado com mil pontos, o segundo com dois mil pontos, o terceiro com três mil pontos e assim sucessivamente até chegar em dez mil pontos. Os intervalos para as coordenadas dos pontos iniciam espaçados em dez mil unidades, no intervalo de -5000 a 5000, e aumentam o intervalo em dez mil unidades até o intervalo ser de cem mil unidades, mantendo a simetria entre números positivos e negativos.

Para cada quantidade de pontos associada a um intervalo específico foi guardado o tempo de execução do algoritmo em segundos. Em seguida foi calculada a média dos tempos de execução do algoritmo para cada quantidade de pontos. Os resultados obtidos nesses experimentos, com o algoritmo de divisão e conquista pode ser visto nas tabelas abaixo:

Tabela 3: As linhas representam as quantidades de pontos e as colunas representam, em unidade de milhar, os intervalos dos pontos gerados.

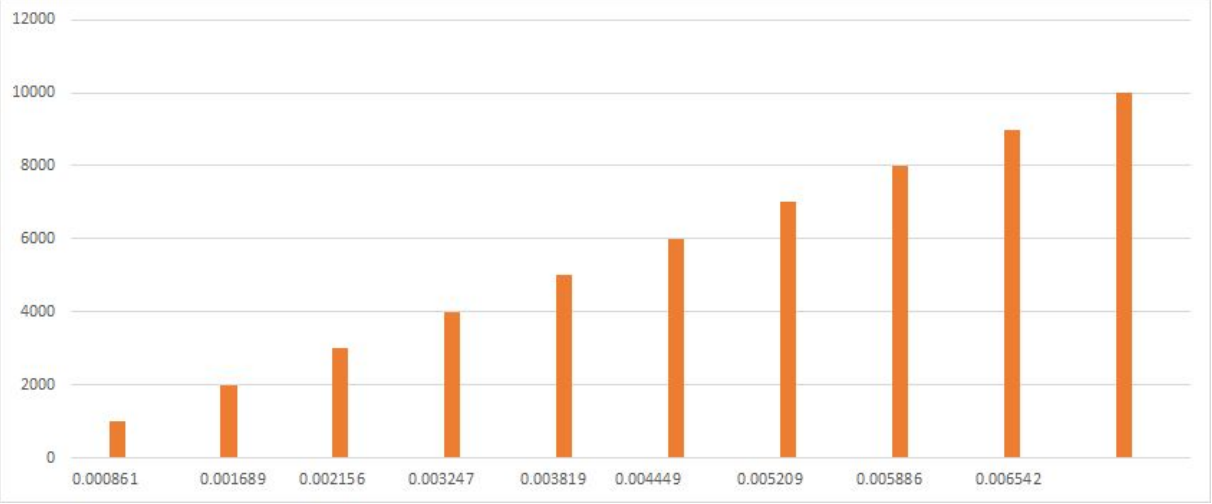
	-5 a 5	**-10 a 10**	**-15 a 15**	**-20 a 20**	**-25 a 25**	**-30 a 30**	**-35 a 35**	**-40 a 40**	**-45 a 45**	**-50 a 50**
1000	0.00109	0.00056	0.00090	0.00079	0.00090	0.00103	0.00089	0.00096	0.00095	0.00054
2000	0.00168	0.00174	0.00173	0.00181	0.00194	0.00162	0.00174	0.00154	0.00146	0.00163
3000	0.00250	0.00247	0.00253	0.00221	0.00261	0.00247	0.00189	0.00244	0.00223	0.00210

4000	0.00359	0.00332	0.00320	0.00306	0.00309	0.00269	0.00306	0.00435	0.00308	0.00303
5000	0.00374	0.00372	0.00384	0.00387	0.00424	0.00379	0.00367	0.00373	0.00361	0.00398
6000	0.00434	0.00465	0.00440	0.00424	0.00501	0.00458	0.00423	0.00426	0.00421	0.00457
7000	0.00562	0.00530	0.00494	0.00515	0.00523	0.00519	0.00526	0.00495	0.00526	0.00519
8000	0.00594	0.00628	0.00602	0.00593	0.00575	0.00591	0.00569	0.00593	0.00531	0.00610
9000	0.00701	0.00640	0.00682	0.00609	0.00645	0.00681	0.00627	0.00649	0.00679	0.00629
10000	0.00856	0.00711	0.00712	0.00720	0.00706	0.00682	0.00696	0.00689	0.00701	0.00688

Tabela 4: Média e intervalo de confiança para cada quantidade de pontos.

Quantidade de Pontos	Média	Intervalo de Confiança
1000	0.000861	0.000748 – 0,000974
2000	0.001689	0.001604 – 0.001774
3000	0.002156	0.001712 – 0.002600
4000	0.003247	0.002968 – 0.003526
5000	0.003819	0.003706 – 0.003932
6000	0.004449	0.004291 – 0.004607
7000	0.005209	0.005091 – 0.005327
8000	0.005886	0.005724 – 0.006048
9000	0.006542	0.006357 – 0.006727
10000	0.007161	0.006847 – 0.007475

Grafico 2: Pontos por média de tempo.



Analogamente a força bruta, após analisar o gráfico é possível perceber que o tempo aumenta de acordo com a quantidade de pontos.

4. Conclusão

Após a realização dos testes, é nítido observar que a melhoria implementada no algoritmo de Força Bruta reduziu o tempo consideravelmente de minutos para segundos nos testes desenvolvidos.

O algoritmo de Divisão e Conquista também foi bem eficiente para os casos testes. Uma melhoria futura a ser trabalhada é diminuir a quantidade de vetores trabalhados na divisão e conquista, pois a cada momento da recursão são criados vetores para guardar os pontos.

5. Referências

R. Rossetti, A.P. Rocha, A. Pereira, P.B. Silva, T. Fernandes (Concepção e Análise de Algoritmos - MIEIC). Disponível em:

https://paginas.fe.up.pt/~rossetti/rrwiki/lib/exe/fetch.php?media=teaching%3A1011%3Acal%3Acalaulapratica03.pdf

Paulo Feofiloff, José Coelho de Pina (Análise de Algoritmos - USP). Disponível em:

https://www.ime.usp.br/~cris/aulas/10_1_6711/slides/aula6.pdf