Trabalho Prático 01 - Projeto e Análise de Algoritmos

Pedro Henrique Reis Rodrigues

Novembro 2021

1 Introdução

Para este trabalho prático foi proposto a solução do problema de determinar a distância mínima entre um ponto ao outro, dado um espaço em 2d com diversos pontos no plano. O método apresentado realiza o cálculo das distâncias entre os pontos através da força bruta, divisão e conquista, e uma versão aprimorada do algoritmo de divisão e conquista.

2 Implementação

2.1 Métodos de resolução do algoritmo

Para tal implementação foi utilizado uma struct `Ponto` que possui apenas um inteiro X e um inteiro Y representando quais são as dimensões desse ponto no espaço. A leitura dos pontos que serão analisados será feita através de um arquivo chamado `"pontos.txt"`.

A primeira linha do arquivo contém a quantidade de pontos que estão no plano e as demais linhas até o final do arquivo contém os pares de coordenadas separados por um espaço simples entre a coordenada X e coordenada Y e uma quebra de linha entre cada par.

Após a leitura da quantidade de pontos, um vetor é alocado estáticamente.Feito a interação com o arquivo, é realizado uma pergunta ao usuário como ele deseja realizar o problema sugerido, utilizando qual método, no caso força bruta, Divisão e conquista, ou Divisão e conquista com uma estratégia eficiente. Vale ressaltar que a complexidade do método de força bruta é `O(n^2)`, sendo assim, o pior dentre os que serão analisados, isso é interessante para realizar uma análise temporal no programa, podendo assim realizar uma grande diferença entre tempos de execução em um caso de teste muito grande.

2.2 Força Bruta

A função força bruta, embora ser a pior dentre as três utilizadas, é o método mais trivial e simples para a resolução, em que é analisado de forma sequencial, ponto à ponto, independente de sua posição ou divisão, se assemelhando bastante à estrutura utililizada no `Bubble Sort`, porém ao invés de ordernar, será analisado em cada intereção a distância entre os pontos, e sempre será armazenado em uma variável `min` a menor distância encontrada até então, e assim, será retornado o resultado.

Sua ordem de complexidade é `O(n^2)` visto que é preciso fazer N análises para N elementos, demandando assim um custo computacional não eficiente.

Abaixo será demonstrado como foi feito os métodos de excecução utilizando divisão e conquista, que por mais que seja um pouco mais complexo do que a força bruta, apresenta uma melhora no custo computacional muito significante.

2.3 Divisão e Conquista

Na função Divisão e Conquista, é primeiramente iniciada com um vetor de pontos, indicando os pontos presentes no plano, assim como na força bruta, porém agora, foi feito uma ordenação deste vetor tendo como base as coordenadas X.

Após isso, se dá inicio realmente ao algoritmo, como o próprio nome do método de resolução já sugere, iremos realizar uma divisão ao meio do `array`, em que será criado uma váriavel `meio` que será `array.size() / 2` para separar a array em duas partes, tendo assim num lado da `subarray P[0] até P[meio]` e `P[meio+1] até P[n-1]`. A cada interação iremos dividir o espaço bidimensional em 2x, até que o tamanho seja menor ou igual a 3 pontos no espaço, partindo assim para a resolução através da `Força Bruta`.

Feito isso, será utilizado um método, por meio de recursividade, para calcular as distâncias entre as `subarray`, de modo que será calculado a distância mínima da `subarray da direita`, sendo armazenada na váriavel `DistD` e também será calculada a menor distância na `subarray da esquerda`, sendo também armazenada numa variável `DistE`.

Tendo em mãos as duas distâncias que são mínimas, será armazenado numa variável `dist` a distância menor entre essas duas `DistD` e `DistE`.

Mas atente-se, que esta ainda não é a resposta, deve-se analisar os pontos que estão próximos à fronteira das duas `subarrays`, em que a distância mínima encontrada será usada como critério de até onde as barreiras da esquerda e direita irão, assim como pode ser visto na figura abaixo:

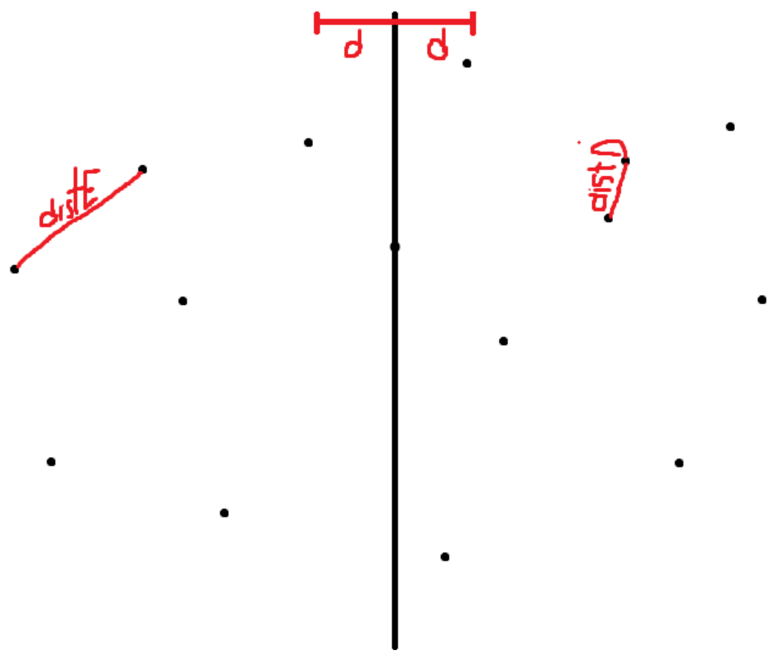


Figura 1: Método para achar os pontos próximos à fronteira

Será armazenado num vetor `strip` todos os pontos que estão dentro dessa margem definida pela `dist`, após isso iremos ordenar este vetor, tendo em comparação agora as coordenadas Y, e após isso será utilizado uma técnica matemática e geogébrica para achar a distância dos pontos tendo ordem de complexidade O(n), isto é, visitando os pontos uma vez apenas.

Para níveis mais didáticos, vamos renomear a distância mínima, que antes era `dist`, para `d`, afim de ajudar a entender os cálculos a serem feitos.

Este método será a parte mais importante e complexa do código, pois o fato é que não precisamos analisar cada ponto do `strip` com todos os outros pontos restantes do `strip`. Em vez disso, consideraremos apenas os pontos que estão a menos de `d` distância daquele ponto. Para isso, temos que considerar um retângulo 2*`d` por `dist` dividido em 8 quadrados (como é mostrado na figura)

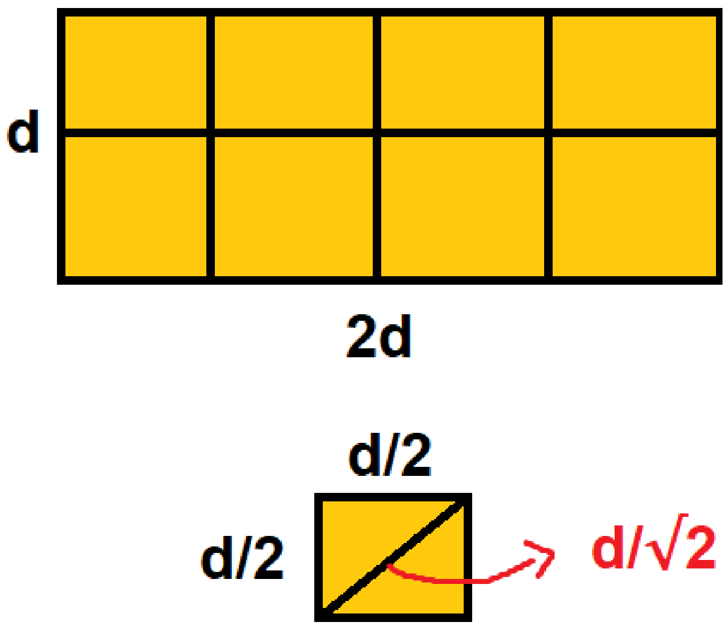


Figura 2: Retângulo dividido em 8 partes (8 quadrados de lado d/2)

Como pode ser visto na figura, todos os lados `d`/2 do quadrado estão de modo em que nenhum quadrado seja compartilhado entre as duas metades, ou seja, um quadrado pertence completamente à esquerda ou completamente à direita da linha. Visto isso, 2 pontos não podem pertencer ao mesmo quadrado `d`/2, porque a distância máxima entre 2 pontos no mesmo quadrado pode ser `d`/2 (comprimento da diagonal) e já sabemos que a distância mínima entre 2 pontos no mesmo quadrado metade é `d` e `d`/2, que é menor que `d`, não é possível.

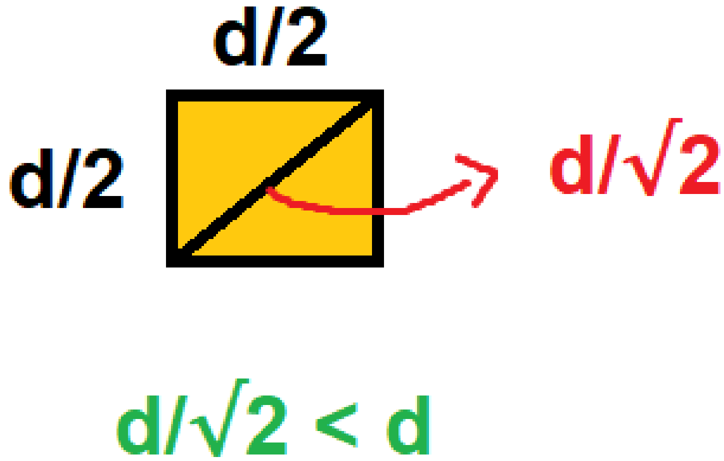


Figura 3: 2 Pontos não podem permanecer no mesmo quadrado

Portanto, 1 ponto pertence a 1 quadrado no máximo e, consequentemente, o retângulo 2`d` por `d` pode ter no máximo 8 pontos e qualquer ponto nesta `strip` precisa ser comparado com os outros 7 pontos. Se tentarmos comparar um ponto a pontos situados fora desse retângulo, então eles certamente terão uma distância maior que `d`, descartanto totalmente essa possibilidade. Portanto, em vez de cada ponto ser comparado a todos os outros pontos da `strip`, estamos dividindo-o em apenas outros 7 pontos no máximo, tornando a complexidade O(7n), para n pontos na `strip` que é o mesmo equivalente a O(n).

Resumidamente, a ideia básica é que cada ponto não precisa ser comparado com o resto dos pontos da `strip`, mas apenas os pontos que podem ter uma distância menor que `d` entre eles.

Por fim, será retornado para o main, o mínimo entre as distâncias encontradas fora do `strip`, e as distâncias encontradas no `strip`.

Calculando sua complexidade, chegaremos que a operação para ordernar será O(n log n), e como são feitas 2 ordenações durante o processo, além de criar o vetor e procurar as distâncias no `strip`, somando assim mais 2 ordens de O(n), sendo assim:

`T(n) = 2T (n/2) + O(n) + O(n log n) + O(n)`

Como a operação de soma entre os O(n) ainda se mantem O(n), e visto que 2T(n/2) equivale a O(log n), teremos

`T(n) = O(n log^2 n)`

O código já demonstrou uma ótima melhoria em relação ao força bruta, porém pode-se ser aprimorado ainda mais, chegando a `O(n log n)`, e isso se deve ao método de ordenação utilizado para ordernar o `strip`, vejamos o que foi feito no próximo método.

2.4 Divisão e Conquista Aprimorado

Na função Divisão e Conquista Aprimorado, segue como base, basicamente todas às premissas já realizadas no método anterior, com apenas uma grande diferença, que fará com que o código se otimize bastante.

Esta diferença está no modo em que é executado a ordenação do vetor `strip`, em que no método anterior, foi feito através de um Quick Sort modificado, e em vez disso, será feito através de um Merge Sort desta vez.

Outra grande diferença, é que no código anterior, fizemos para cada interação no vetor `strip`, uma ordenação em relação às coordenadas Y, neste código aprimorado, faremos uma ordenação prévia de todos os pontos, tanto em coordenadas Y como coordenadas X, chamaremos esses vetores com os pontos ordenados de `Py[]` e `Px[]`.

Diferentemente da anterior, desta vez iremos utilizar um vetor chamado `Py`, que armazenará todos os pontos, de forma ordenada em relação à coordenada Y, e quando realizarmos as chamadas recursivas, armazenaremos sempre as divisões do plano de maneira ordenada em relação à Y, chamaremos os vetores responsáveis por armazenarem as divisões de `Pyd` para os pontos da direita e `Pye` para os pontos da esquerda.

Por fim, será realizado a mesma maneira de calcular os pontos próximos das fronteiras, utilizaremos a mesma ideologia já explicada no tópico acima, em que se dá a entender que estamos realizando um loop com complexidade de `O(n^2)`, porém serão apenas no máximo 6 vezes em que o código será rodado.

A ordem de complexidade no final será melhor do que a divisão e conquista, e isso podemos explicar tanto teoricamente, quanto na prática, já que foi realizado testes com arquivos com mais de 5000 pontos, ambos os mesmos pontos foram calculados utilizando as diferentes 3 técnicas de resolução, e mais abaixo será mostrado os resultados. Portanto, sua ordem de complexidade ficou

`T(n) = 2T (n/2) + O(n) + O(n) + O(n)`

Como a operação de soma entre os O(n) ainda se mantem O(n), e visto que 2T(n/2) equivale a O(log n), teremos no final

`T(n) = O(n log n)`

3 Código fonte

```
#include <stdio.h>
#include <stdlib.h>
#include <vector>
#include <iomanip>
#include <iostream>
#include <cstdio>
#include <cstring>
#include <math.h>
#include <vector>
#include "sys/time.h"
```

```
using namespace std;
#define INF 0x3f3f3f3f

// ALUNO: PEDRO HENRIQUE REIS RODRIGUES
// MATRCULA: 668443
// PROFESSOR: SILVIO GUIMARAES
// TRABALHO: CLOSEST PAIR OF POINTS
// Complexidade O(n log n) em que n representa o n mero de pontos a serem anali

// M todos utilizados: DIVIS O E CONQUISTA e FOR A BRUTA

typedef struct Ponto{
    int x;
    int y;
} Ponto;

int compararX (const void *x, const void *y) {
    Ponto *a = (Ponto*)x;
    Ponto *b = (Ponto*)y;
    return (a->x != b->x) ? (a->x - b->x) : (a->y - b->y);
}

int compararY (const void *x, const void *y) {
    Ponto *a = (Ponto*)x;
    Ponto *b = (Ponto*)y;
    return (a->y != b->y) ? (a->y - b->y) : (a->y - b->y);
}

float distancia(Ponto X, Ponto Y){
    return sqrt((pow(X.x - Y.x,2)+pow(X.y-Y.y,2)));
}

float forca_bruta(Ponto P[],int N){

    float min = INF;
    for(int i = 0 ; i < N ; i++){
        for(int j= i+1; j< N; j++){
            if(distancia(P[i],P[j]) < min){
                min = distancia(P[i],P[j]);
            }
        }
    }
    return min;
}// O(n^2)
```

```
float divisao_conquista(Ponto P[], int N){

    if (N<=3){
        return forca_bruta(P,N);
    }

    int meio = N/2;
    Ponto Pm = P[meio];

    float distE = divisao_conquista(P,meio);
    float distD = divisao_conquista(P+meio,N-meio);
    float dist;

    if(distE <= distD){
        dist = distE;
    }
    else{
        dist = distD;
    }


    Ponto strip[N];
    int j = 0;

    for(int i = 0; i< N; i++){
        if(abs(P[i].x - Pm.x) < dist){
            strip[j]=P[i];
            j++;
        }
    }

    float min = dist;
    qsort(strip,j,sizeof(Ponto),compararY);

    for(int i = 0; i< j;i++){
        for(int k = i+1; k< j && (strip[k].y-strip[i].y)<min ; k++){
            if(distancia(strip[i],strip[k])<min){
                min = distancia(strip[i],strip[k]);
            }
        }
    }

    return min;

} // O(n log^2 n)
```

```
float div_conquista(Ponto P[], int N){

    qsort(P,N,sizeof(Ponto), compararX);
    return divisao_conquista(P,N);

}

float divisao_conquista_apri(Ponto Px[], Ponto Py[], int N){

    if(N <= 3){
        return forca_bruta(Px,N);
    }

    int meio = N/2;

    Ponto Pm = Px[meio];
    Ponto Pye[meio];
    Ponto Pyd[N-meio];
    int aux = 0, aux1 = 0;
    for(int i = 0 ; i < N ; i++){
        if(Py[i].x < Pm.x || (Py[i].x == Pm.x && Py[i].y == Pm.y) && aux < meio)
            Pye[aux++] = Py[i];
        }
        else{
            Pyd[aux1++]= Py[i];
        }
    }
    float distE = divisao_conquista_apri(Px,Pye,meio);
    float distD = divisao_conquista_apri(Px + meio, Pyd, N-meio);
    float dist;

    if(distE <= distD){
        dist = distE;
    }
    else{
        dist = distD;
    }

    Ponto strip[N];
    int j = 0;
    for(int i = 0 ; i< N ; i++){
        if(abs(Py[i].x - Pm.x)< dist){
            strip[j] = Py[i];
            j++;
        }
    }
```

```

    float min = dist;
    for(int i = 0 ; i < j; i++){
        for(int k = i+1; k < j && (strip[k].y - strip[i].y) <min ; k++){
            if(distancia(strip[i],strip[k])<min){
                min = distancia(strip[i],strip[k]);
            }
        }
    }
    return min;
} // O(n log n)

float div_conquista_apri(Ponto P[], int N){
    Ponto Px[N];
    Ponto Py[N];
    for(int i = 0; i< N ; i++){
        Px[i] = P[i];
        Py[i] = P[i];
    }
    qsort(Px,N,sizeof(Ponto), compararX);
    qsort(Py,N,sizeof(Ponto),compararY);
    return divisao_conquista_apri(Px,Py,N);
}

int main(){
    int N;
    FILE *fp;
    fp = fopen("pontos.txt","r");
    fscanf(fp,"%d\n",&N);
    printf("%d",N);
    Ponto P[N];
    for(int i = 0 ; i< N; i++){
        fscanf(fp,"%d %d\n",&P[i].x,&P[i].y);
    }
    fclose(fp);

    struct timeval start,end;
    double tempo;
    int choice;
    do{

        printf("\n\n\n\nResolver por:\n[1] O(n^2) - Forca Bruta\n[2] O(nlog^2n) 
        scanf("%d",&choice);
        switch (choice){
            case 0:
                break;
```

```
            case 1:
                gettimeofday(&start,NULL);
                printf("Distancia: %f",forca_bruta(P,N));
                gettimeofday(&end, NULL);

                tempo = (end.tv_sec - start.tv_sec)*1e6;
                tempo = (tempo + (end.tv_usec-start.tv_usec))*1e-6;
                printf("\nTempo de execucao: %lf segundos",tempo);
                break;
            case 2:
                gettimeofday(&start,NULL);
                printf("Distancia: %f",div_conquista(P,N));
                gettimeofday(&end, NULL);

                tempo = (end.tv_sec - start.tv_sec)*1e6;
                tempo = (tempo + (end.tv_usec-start.tv_usec))*1e-6;
                printf("\nTempo de execucao: %lf segundos",tempo);
                break;
            case 3:
                gettimeofday(&start,NULL);
                printf("Distancia: %f",div_conquista_apri(P,N));
                gettimeofday(&end, NULL);

                tempo = (end.tv_sec - start.tv_sec)*1e6;
                tempo = (tempo + (end.tv_usec-start.tv_usec))*1e-6;
                printf("\nTempo de execucao: %lf segundos",tempo);
                break;
            default:
                printf("Erro na inicializacao");
        }

    }while(choice != 0);

    return 0;

}
```

O código fonte enviado juntamente com este documento está devidamente comentado nas partes importantes do código.

4 Experimentos e resultados obtidos

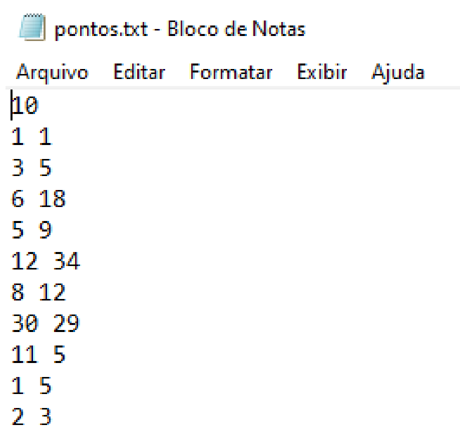
4.1 Arquivo utilizado

Para verificar se o algoritmo está funcionando, primeiramente foi analisado um arquivo possuindo 10 pontos para verificar a funcionabilidade, e assim, foi possível prosseguir para arquivos superiores.

Para o experimento mais robusto, foi utilizado um arquivo gerado automaticamente por um programa feito na linguagem C, em que escreve no arquivo, um número de nossa escolha, que será o número de pontos que queremos criar, e o algoritmo cria aleatóriamente as coordenadas X e Y, separadas de um espaço, a cada ponto há uma quebra de linha, indicando que o outro ponto começa a partir de lá.

4.2 Experimentos e resultados obtidos

Verificando a funcionabilidade, foi utilizado este arquivo com 10 pontos


pontos.txt - Bloco de Notas

Arquivo Editar Formatar Exibir Ajuda

```
10
1 1
3 5
6 18
5 9
12 34
8 12
30 29
11 5
1 5
2 3
```

Figura 4: Arquivo com 10 pontos

Em que calculando anteriormente, o resultado seria de 2.0 metros de distância. Abaixo estão as outputs dos teste realizados utilizando o método de força bruta, e depois o divisão e conquista, e por fim o divisão e conquista aprimorado, respectivamente.

```
Resolver por:
[1] O(n^2) - Forca Bruta
[2] O(nlog^2n) - Divisao e Conquista
[3] O(nlogn) - Divisao e Conquista otimizado
[0] Encerrar o programa
1
Distancia: 2.000000
Tempo de execucao: 0.000000 segundos
```

Figura 5: Output Força Bruta com Arquivo com 10 pontos

```
Resolver por:
[1] O(n^2) - Forca Bruta
[2] O(nlog^2n) - Divisao e Conquista
[3] O(nlogn) - Divisao e Conquista otimizado
[0] Encerrar o programa
2
Distancia: 2.000000
Tempo de execucao: 0.000000 segundos
```

Figura 6: Output Divisão e Conquista com Arquivo com 10 pontos

```
Resolver por:
[1] O(n^2) - Forca Bruta
[2] O(nlog^2n) - Divisao e Conquista
[3] O(nlogn) - Divisao e Conquista otimizado
[0] Encerrar o programa
3
Distancia: 2.000000
Tempo de execucao: 0.000000 segundos
```

Figura 7: Output Divisão e Conquista aprimorado com Arquivo com 10 pontos

Pode-se notar que ambos os 3 métodos responderam de forma corretamente, porém, não foi possível enxergar de perto a diferença de desempenho, por se tratar de um arquivo muito pequeno. Portanto foi utilizado um arquivo com

5000 pontos para testes mais robustos, e os pontos foram criados de forma aleatória. E estes são os resultados:

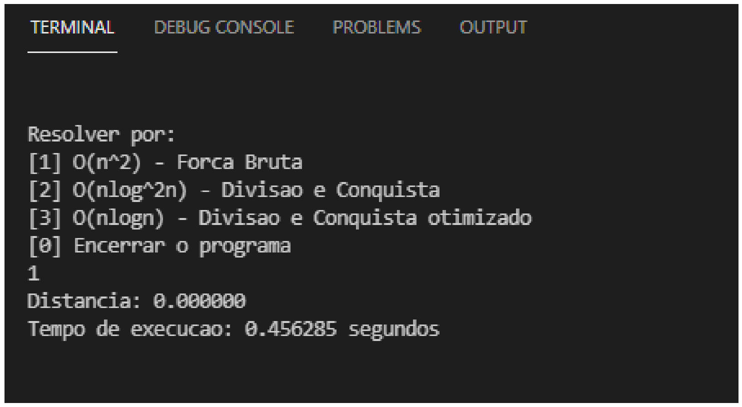


Figura 8: Output Força Bruta com Arquivo com 5000 pontos

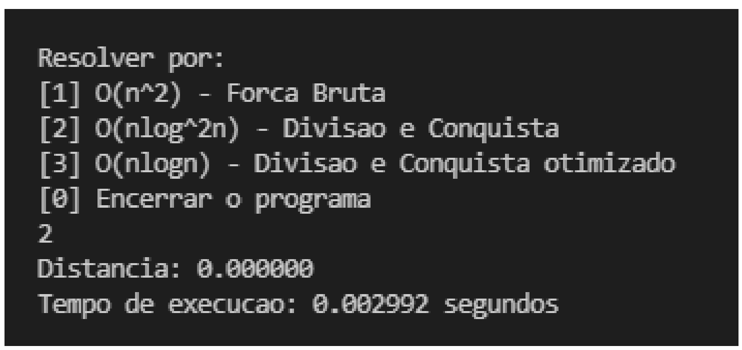


Figura 9: Output Divisão e Conquista com Arquivo com 5000 pontos

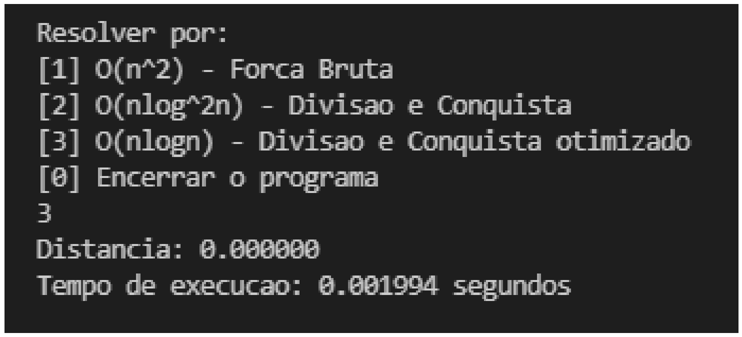


Figura 10: Output Divisão e Conquista aprimorado com Arquivo com 5000 pontos

5 Conclusão

Finalizando o teste com 5000 pontos, foi possível afirmar que a distância mínima entre os pontos era igual a 0, ou seja, existia algum ponto que estava sobreposto à outro, porém isso não é o que estamos querendo observar com esse teste, já que ele foi criado especificamente para ver o desempenho dos nossos algoritmos criados. Portanto, graças ao nosso arquivo teste, podemos concluir o quão eficiente nosso código utilizando divisão e conquista ficou comparado ao algoritmo trivial que seria através da força bruta, isso se deve pois este tipo de problema é de extrema necessidade utilizar uma técnica que divida os problemas maiores em problemas menores, uma vez que neste caso, os problemas menores são independentes um aos outros, sendo assim será mais rápido a resolução destes, e por fim, a combinação destes pequenos problemas gerará para nós o resultado do problema maior.

Referências

[1] Shamos, Michael Ian; Hoey, Dan (1975). "Closest-point problems". 16th Annual Symposium on Foundations of Computer Science, Berkeley, California, USA, October 13-15, 1975. IEEE Computer Society. pp. 151–162.

[2] Rabin, M. (1976). "Probabilistic algorithms". Algorithms and Complexity: Recent Results and New Directions. Academic Press. pp. 21–39. As cited by Khuller Matias (1995).

[3] Fortune, Steve; Hopcroft, John (1979). "A note on Rabin's nearest-neighbor algorithm". Information Processing Letters. 8 (1): 20–23.

[4] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001) [1990]. "33.4: Finding the closest pair of points". Introduction to Algorithms (2nd ed.). MIT Press and McGraw-Hill. pp. 957–961.

[5] http://people.csail.mit.edu/indyk/6.838-old/handouts/lec17.pdf