Análise de Soluções Computacionais para o Problema do Par Mais Próximo

Breno M. de Abreu

Bacharelado em Sistemas de Informação - Universidade Tecnológica Federal do Paraná

brenoabreu@utfpr.alunos.edu.br

1. Resumo

Este trabalho visa encontrar uma solução computacional para o problema do par mais próximo de pontos em um plano em distância euclidiana. Foi possível solucionar o problema utilizando os algoritmos força bruta e divisão e conquista, o primeiro com complexidade $O(n^2)$ e o segundo com $\Theta(n \log n)$. Apesar da superioridade do segundo tratando-se de velocidade de execução, o algoritmo se mostrou extenso e complexo, sendo necessário escrever uma quantidade substancialmente maior de código para realizá-lo. Além disso, para tamanhos de entradas menores que aproximadamente 45 pontos, o algoritmo força bruta se mostrou mais rápido, ainda que a diferença seja baixa. Para valores maiores a diferença de velocidade é consideravelmente maior, sendo a divisão e conquista mais rápida.

2. Algoritmo Força Bruta

O algoritmo força bruta resolve o problema do par mais próximo comparando cada ponto com todos os outros e funciona da seguinte forma: para cada elemento do vetor - menos o último - realiza o cálculo de distância euclidiana com todos os elementos posteriores à ele. Enquanto compara, uma variável que recebe a distância do par mais próximo é atualizada caso uma distância encontrada seja menor que a distância armazenada na variável.

O pseudocódigo a seguir descreve a função:

Procedimento Força Bruta (V)
Entrada
V - vetor de pontos no plano bidimensional
Saída
minDist - menor distância entre dois pontos presentes no vetor V
Início
1 minDist ← INFINITO
2 **Para** i **de** 0 **até** V.tamanho - 1 **faça**
3 **Para** j **de** i + 1 **até** V.tamanho **faça**
4 **Se** distância (V[i], V[j]) < minDist **então**
5 minDist ← distância (V[i], V[j])
6 **Fim-Se**
7 **Fim-Para**
8 **Fim-Para**
9 **Retorne** minDist
Fim Força Bruta

Como é possível perceber, o primeiro laço percorre todos os elementos, o segundo, dentro do primeiro, percorre todos os elementos posteriores resultando em um algoritmo de complexidade $O(n^2)$.

3. Algoritmo Divisão e Conquista

O problema ainda pode ser solucionado utilizando a técnica de divisão e conquista que possibilita reduzir o tempo de execução da operação substancialmente. Uma possível solução está em criar um algoritmo com complexidade $\Theta(n\ (\log n)^2)$[1], porém, foi possível encontrar uma solução ainda melhor com complexidade $\Theta(n \log n)$[2].

A implementação do algoritmo em questão apresenta pequenas diferenças dependendo do autor. Por exemplo, enquanto que a implementação encontrada no artigo do site GeeksForGeeks[2] determina a linha vertical divisória como sendo a coordenada x do ponto que está no meio do vetor, o algoritmo implementado por Hadzilacos e Toueg[3] define a mesma linha como a média aritmética entre a coordenada x do último ponto no vetor da primeira metade do vetor principal, e o primeiro ponto no vetor da segunda metade do vetor principal. Outra diferença está na quantidade de pontos subsequentes quando o algoritmo tenta encontrar a menor distância entre pontos presentes na faixa divisória. Enquanto algumas fontes indicam que o valor máximo deve ser 15 [6][7], outras indicam que o valor deve ser 7 [3][5]. Dessa forma, o algoritmo descrito neste documento se utiliza das técnicas mais apropriadas julgadas pelo autor, que serão justificadas nos tópicos posteriores.

3.1 Algoritmo

Antes do algoritmo em si ser executado é necessário criar dois vetores a partir do vetor principal de pontos: um contendo os pontos ordenados por ordem de sua coordenada x, e outro por sua coordenada y. Tais vetores serão chamados de *PX* e *PY* respectivamente. Estes dois vetores ordenados então são enviados para para o algoritmo principal, que será chamado recursivamente até o resultado ser encontrado.

Inicialmente, para a etapa de divisão, é necessário criar uma linha imaginária vertical que separa os pontos na metade. O valor da coordenada x que representa esta linha é encontrado executando as seguintes operações: primeiro encontra-se o índice central do vetor *PX*; depois tira-se a média aritmética entre o ponto central encontrado a partir do índice descoberto anteriormente e o ponto subsequente a este utilizando a coordenada x.

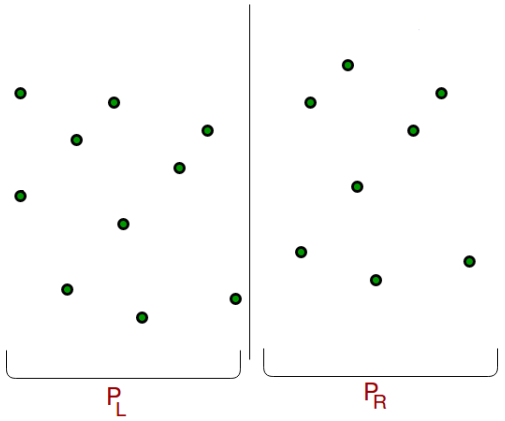

Figura 1. Divisão do vetor principal em duas partes

Por isso é necessário que o vetor *PX* esteja ordenado previamente, para que seja possível dividi-lo na metade de acordo com sua coordenada x. Assim, os pontos que possuem a coordenada x menor que a linha vertical ficam à esquerda da linha (PL na figura 1), agrupados em seu próprio vetor, e os demais à direita da linha (PR na figura 1), também agrupados em outro vetor. Estes dois vetores resultantes serão chamados *PXE* e *PXD* respectivamente.

Ainda é necessário criar outros dois vetores, desta vez utilizando como vetor principal *PY*. O processo consiste em agrupar os pontos presentes em *PY* de acordo com sua localização em relação à linha vertical divisória. Caso sua coordenada x seja menor que o valor da linha vertical, o ponto é inserido em um determinado vetor, caso contrário, é inserido em outro. Os vetores resultantes serão chamados de *PYE* e *PYD* respectivamente. Como o vetor *PY* já está ordenado, há garantia de que os dois vetores resultantes também serão ordenados de acordo com sua coordenada y.

Resumindo: através dos processos citados anteriormente, são criados quatro novos vetores a partir dos dois vetores principais recebidos pela função. Um deles armazena os pontos à esquerda da linha vertical divisória ordenados pela coordenada x, outro contém os

mesmos pontos mas ordenados pela coordenada y, há ainda um terceiro vetor contendo os pontos que estão à direita da linha ordenados pela coordenada x, e o último contendo estes mesmos pontos ordenados pela coordenada y.

Os vetores resultantes são então enviados recursivamente para a mesma função em duas chamadas. A primeira recebe *PXE* e *PYE*, e a segunda *PXD* e *PYD*, caracterizando o processo de divisão do algoritmo divisão e conquista.

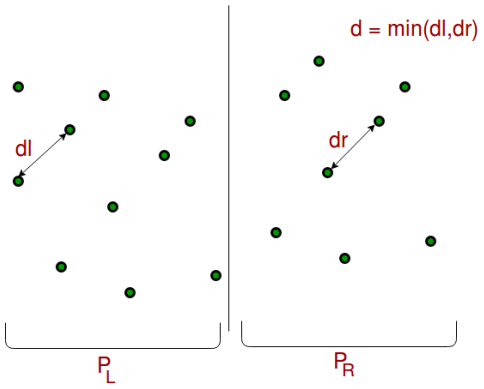

Figura 2. Par mais próximo em cada subvetor

Para dar início ao processo de conquista, quando os subvetores se tornam pequenos o suficiente, isto é, contendo três ou menos elementos, a menor distância entre dois pontos é encontrada utilizando a técnica força bruta. O resultado então é enviado para as camadas de cima do processo recursivo.

Quando a menor distância é encontrada nos subvetores *PXE* e *PXD* (par de pontos dl e dr respectivamente na figura 2), uma comparação entre estas é realizada para encontrar qual dos pares encontrados possui a menor distância (variável d na figura 2). Esta distância será chamada de *dist*.

Com a etapa anterior foi encontrado o par de pontos com a menor distância entre os subvetores à esquerda e à direita da linha divisória, porém, a comparação foi realizada apenas entre os pontos presentes em cada um desses subvetores e não leva em conta a distância entre pontos presentes antes e após a linha. Dessa forma, não é possível saber se a menor distância está entre um ponto presente em *PXE* e outro presente em *PXD*.

Para encontrar a distância entre pontos presentes em vetores diferentes é necessário criar uma faixa vertical imaginária com largura 2 * *dist* (faixa S na figura 3), sendo que seu centro é a linha vertical divisória encontrada anteriormente.

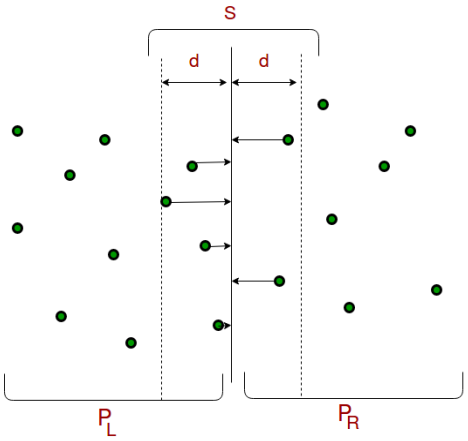

Figura 3. Faixa com largura 2 * *dist* entre os subvetores

O valor 2 * *dist* para a largura é utilizado pois como sabe-se que a menor distância entre pontos encontrada até agora foi *dist*, não é possível que um par de pontos exceda essa distância caso tenham o mesmo valor da coordenada y. Por exemplo, suponhamos que um ponto esteja bem acima da linha vertical. Como sabemos que a menor distância entre pontos presentes no mesmo vetor já foi descoberta, descartamos a possibilidade de haver uma distância menor entre esse ponto e outro que esteja no mesmo lado, assim, comparamos o ponto com aqueles presentes no lado oposto. Como a maior distância horizontal possível é *dist*, e o ponto está bem acima da linha vertical, pode-se descartar todos os pontos que estejam a uma distância maior que *dist* da linha vertical, pois todos eles certamente também estão a uma distância maior que *dist* do ponto aqui imaginado.

Cria-se então um novo vetor, denominado aqui *PF*, que irá conter todos os pontos que estão a uma distância menor que *dist* da linha vertical. Este vetor é gerado a partir de *PY* para que os pontos estejam ordenados de acordo com a coordenada y.

A próxima etapa consistiria em comparar um ponto com todos os pontos subsequentes assim como no algoritmo força bruta, porém tal ação pode ser executada apenas para os 7 pontos subsequentes à um ponto. O que seria um algoritmo com complexidade quadrática agora pode ser resolvido linearmente. A prova consiste em criar uma grade imaginária como apresentado na figura 4, em que a linha central indica a linha vertical divisória. A grade foi criada de forma que a linha mais abaixo esteja na mesma coordenada y do ponto que irá ser comparado. Dessa forma, o topo da grade é a distância máxima vertical do ponto pois cada bloco da grade possui *dist*/2 de lado, não sendo necessário comparar pontos que estejam além do topo da grade. Como o vetor está ordenado só é necessário comparar o um ponto com seus pontos subsequentes, por isso o gráfico mostra apenas blocos que estão acima do ponto. É importante perceber que há a possibilidade de haver apenas um ponto em cada um dos blocos, pois a distância máxima entre dois pontos dentro de um bloco é menor que *dist*, enquanto que a maior distância entre dois pontos presentes em dois blocos vizinhos é maior que *dist*. Unindo a ideia que há apenas um ponto por bloco, e que o teto da grade é a maior distância possível do ponto verticalmente, é possível concluir que é necessário apenas comparar um ponto com seus sete pontos subsequentes, já que qualquer ponto acima disso estará certamente a uma distância maior que *dist*.

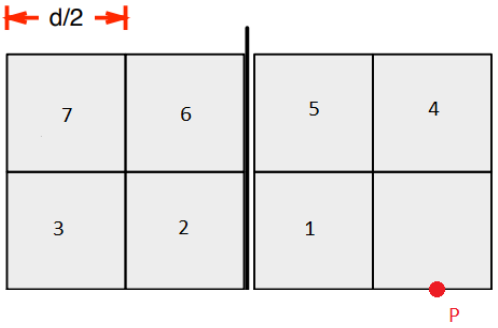

Figura 4. Grade imaginária com blocos de *dist*/2 de lado criada a partir de um ponto

Após encontrar o par de pontos com menor distância presente em *PF,* comparamos o resultado com *dist*. Caso a distância seja menor enviamos o par de pontos encontrado para o nível acima da recursão, caso contrário, enviamos o par de pontos encontrado anteriormente na comparação entre *PXD* e *PXE*.

3.2 Análise de Complexidade Tempo

Para auxiliar a análise de complexidade será utilizado o pseudocódigo do algoritmo divisão e conquista apresentado a seguir:

Procedimento DivisãoConquista (PX, PY)
Entrada
PX - vetor de pontos ordenados pela coordenada x.
PY - vetor de pontos ordenados pela coordenada y.
Saída
parPontos - par de pontos que possui a menor distância entre os pontos presentes no vetor PX.

Início
01 **Se** *PX.tamanho* <= 3 **Então**
02 retorna ForçaBruta(PX)
03 **Se Não**
04 *meio* ← índice central do vetor *PX*
05 *linhaVertical* ← (*PX*[meio] + *PX*[meio + 1]) / 2
06 *PYE* ← subvetor de *PY* com pontos tendo a coordenada x menor que *linhaVertical*
07 *PYD* ← subvetor de *PY* com pontos tendo a coordenada x maior que *linhaVertical*
08 *PXE* ← subvetor contendo os pontos pertencentes a primeira metade de *PX*
09 *PXD* ← subvetor contendo os pontos pertencentes a segunda metade de *PX*
10 *resEsq* ← DivisãoConquista(*PXE*, *PYE*)
11 *resDir* ← DivisãoConquista(*PXD*, *PYD*)
12 *minEsqDir* ← *resEsq* se este tem uma distância menor que *resDir*; se não, *resDir*
13 *PF* ← subvetor de *PY* contendo os pontos que estão a uma distância menor que *minEsqDir.distância* da *linhaVertical* em relação a coordenada x
14 *auxDist* ← INFINITO
14 **Para cada** *ponto* ∈ *PF* **Faça**
15 **Para cada** um dos próximos 7 pontos (*ps*) subsequentes a *ponto* **Faça**
16 **Se** a distância entre *ps* e *ponto* for menor que *auxDist* **Então**
17 *auxDist* ← distância entre *ps* e *ponto*
18 *resFaixa* ← par de pontos *ps* e *ponto*
19 **Fim-Se**
20 **Fim-Para**
21 **Fim-Para**
22 *parPontos* ← *minEsqDir* se este tem uma distância menor que *resFaixa*; se não, *resFaixa*
23 retorna *parPontos*
24 **Fim-Se**
Fim DivisãoConquista

Através da análise do pseudocódigo podemos perceber que a relação de recorrência do algoritmo divisão e conquista é $T(n) = 2T(n/2) + n$.

Para a primeira parte da relação, sabe-se que para a recursão dividi-se os vetores pela metade (linhas 06 à 09), e chama-se recursivamente duas vezes a função *DivisãoConquista* (linhas 10 e 11), uma vez para cada metade. Logo chega-se no resultado $2T(n/2)$.

Para os demais passos, percebe-se que a linha 02 tem complexidade $O(1)$ já que haverá sempre no máximo 3 comparações para encontrar o par de pontos com menor distância usando força bruta independentemente do tamanho do vetor inicial. A seguir as linhas 04 e 05 também possuem complexidade $O(1)$ já que são operações que não dependem da quantidade de elementos dos vetores recebidos. As linhas 06 à 09 podem ser resolvidas percorrendo apenas uma vez os vetores PX e PY para popular os subvetores, logo, sua complexidade é $O(n)$. A linha 12 é uma operação única, tendo complexidade $O(1)$. Na linha 13, a operação pode ser resolvida linearmente percorrendo apenas uma vez o vetor PY, logo, sua complexidade é $O(n)$. Para o algoritmo descrito nas linhas 14 à 21 podemos perceber que por mais que haja um laço dentro de outro, a complexidade é linear, pois o primeiro laço percorre todos os elementos do vetor, tendo complexidade $O(n)$, mas o laço interno da linha 15 tem um número constante de iterações, tendo complexidade $O(1)$. A linha 22 é uma operação executada uma única vez, tendo complexidade $O(1)$.

Com isso, pode-se concluir que a primeira parte da relação de recorrência é $2T(n/2)$, e a segunda é $O(n)$ já que não houve uma complexidade maior encontrada nas etapas de divisão e combinação, levando ao resultado $T(n) = 2T(n/2) + n$.

Pela relação de recorrência descoberta anteriormente temos que: $a = 2$, $b = 2$ e $k = 1$. Através do Teorema Mestre, sabemos que se: $a = b^k$ então $T(n) = \Theta(n^k \log n)$, logo, a complexidade do algoritmo divisão e conquista aqui descrito é $\Theta(n \log n)$.

Importante observar que a complexidade da etapa anterior ao algoritmo descrito - a ordenação dos vetores PX e PY - não foi desenvolvida, isso por que há uma variedade de algoritmos de ordenação com complexidades diferentes, e dependendo do algoritmo escolhido a complexidade resultante seria diferente. Por essa razão preferiu-se discutir a complexidade do algoritmo principal apenas.

4. Experimentos

Foram realizados experimentos utilizando diferentes quantidades de entradas, e com o auxílio de um *script* em Python foi possível gerar gráficos a partir dos resultados obtidos.

Para se ter uma visão geral da eficiência dos algoritmos, primeiramente o programa foi executado 500 vezes, sendo que a cada iteração o número de entradas era adicionado em 10, dessa forma obteve-se a comparação entre os algoritmos com o tamanho do vetor variando de 1 até 5000.

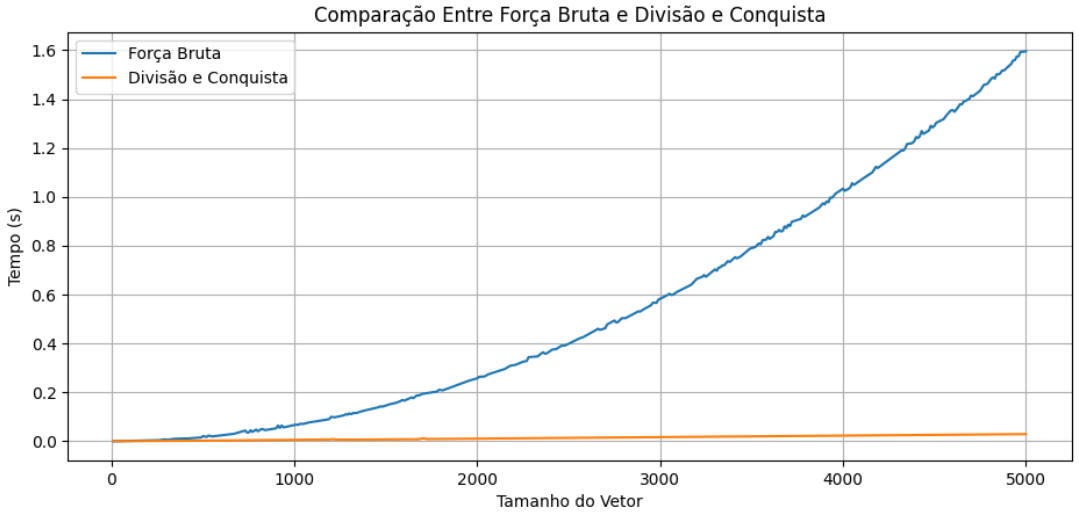

Figura 5. Comparação entre os algoritmos com tamanho do vetor variando de 1 até 5000 entradas

Como é possível observar (Figura 5), o tempo de execução do algoritmo divisão e conquista se manteve estável em qualquer tamanho do vetor, sempre menor que 0.1 segundos, enquanto que o algoritmo força bruta teve seu tempo de execução aumentado consideravelmente quanto maior a quantidade de entradas. Ainda é possível ver que a partir de aproximadamente 4000 entradas o algoritmo força bruta leva mais que um segundo para ser executado. Ainda observamos que para um tamanho do vetor menor que 1000 os dois algoritmos são executados em menos de 0.1 segundos.

Para que fosse possível observar o comportamento dos algoritmos com entradas relativamente grandes, o programa foi executado 16 vezes, sendo que a cada iteração o número de entradas era 2^i sendo i o número da iteração, mostrando a comparação dos algoritmos com um tamanho do vetor variando de 2 até 65536 entradas.

Neste experimento é possível notar (Figura 6), que para entradas muito grandes o tempo de processamento chega a minutos, enquanto que o algoritmo divisão e conquista permanece na casa dos segundos.

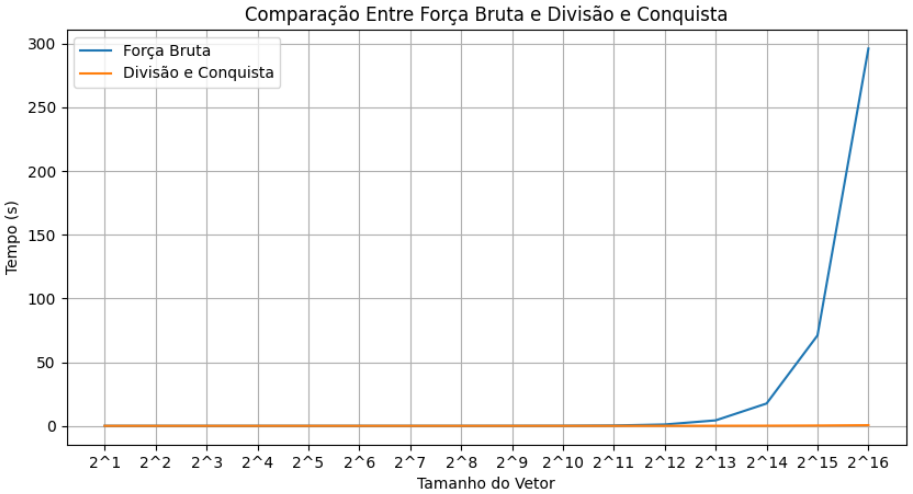

Figura 6. Comparação entre os algoritmos com tamanho do vetor variando de 2 até 65536 entradas

Para observar o comportamento dos algoritmos com uma quantidade de entradas muito pequena, o programa foi executado 100.000 vezes, sendo que a cada iteração o número de entradas era acrescido em 1. O programa executou tamanhos de vetor de 1 até 100 apenas, mas executando a mesma quantidade de entradas 1.000 vezes para então tirar sua média aritmética.

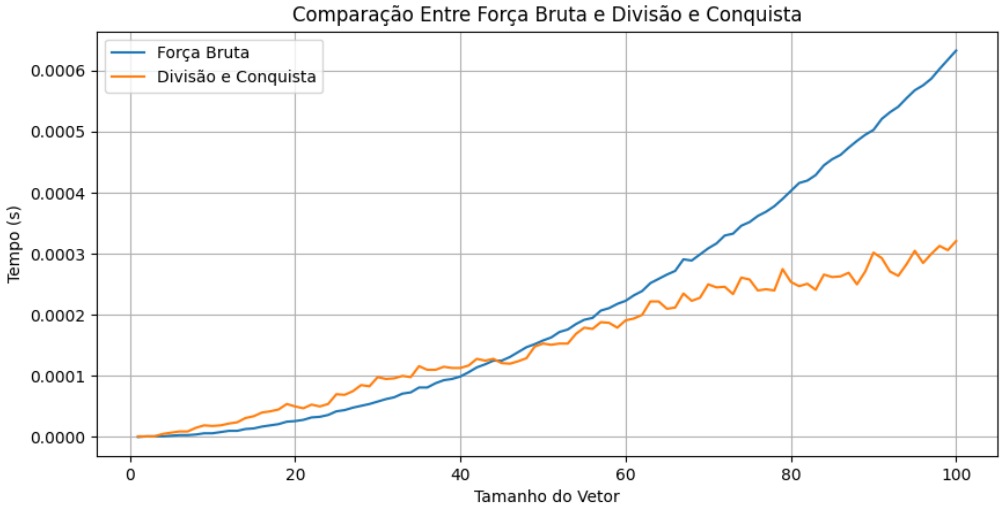

Figura 7. Comparação entre os algoritmos com tamanho do vetor variando de 1 até 100 entradas

Neste experimento é possível observar (Figura 7), que para uma quantidade de entradas menor que aproximadamente 45, o algoritmo força bruta é mais rápido que o algoritmo divisão e conquista.

5. Conclusão

Como comprovado através de experimentos, o algoritmo de divisão e conquista é consideravelmente mais eficiente que o de força bruta, sendo que o primeiro apresenta complexidade $\Theta(n \log n)$ e o segundo $O(n^2)$. Porém, para um número de entradas menor que 45 é possível perceber que o algoritmo força bruta se sobressai, mesmo que não substancialmente, e para entradas relativamente pequenas (na casa das centenas) a diferença de tempo de execução é quase imperceptível. Ainda assim, quanto maior a quantidade de entradas maior é a diferença de tempo entre os dois algoritmos, enquanto o de divisão e conquista se mostra quase estável, demorando no máximo poucos segundos, seja qual for a quantidade de entradas, o de força bruta apresenta resultados que podem levar minutos até encontrar o conjunto de pontos esperado.

Mesmo com a superioridade em questão de tempo de execução do algoritmo de divisão e conquista, este ainda apresenta pontos negativos. Primeiramente há a necessidade dos vetores que serão introduzidos no algoritmo estarem ordenados, o que resulta em mais tempo de execução antes do algoritmo principal ser executado. Além disso, este algoritmo ainda precisa alocar memória extra para poder ser executado, algo que não ocorre no algoritmo força bruta. E por fim, para esse algoritmo ser implementado é necessário escrever uma quantidade maior de linhas de código que seu concorrente.

6. Referências

As figuras 1, 2 e 3 foram retiradas do site GeeksForGeeks[1]. A figura 1 foi alterada para se ajustar à necessidade do autor.

[1] (2020) “Closest Pair of Points using Divide and Conquer algorithm” https://www.geeksforgeeks.org/closest-pair-of-points-using-divide-and-conquer-algorithm/, em: GeeksForGeeks.

[2] (2020) “Closest Pair of Points | O(nlogn) Implementation” https://www.geeksforgeeks.org/closest-pair-of-points-onlogn-implementation/, em: GeeksForGeeks.

[3] Hadzilacos, V. e Toueg, S. (2020) “Algorithm to Find the Closest Pair of Points” http://www.cs.toronto.edu/~vassos/teaching/c73/handouts/closest-pair-of-points.pdf, Computer Science S73, University of Toronto.

[4] (2014) “O(n log n) Algorithm for Closest Pair - 1 | Algorithm”, https://www.youtube.com/watch?v=jAigdwcATNw&ab_channel=FreeEngineeringLectures, em: Free Engineering Lectures, YouTube.

[5] (2014) “O(n log n) Algorithm for Closest Pair - 2 | Algorithm”, https://www.youtube.com/watch?v=PerYilqF6o&ab_channel=FreeEngineeringLectures, em: Free Engineering Lectures, YouTube.

[6] Carl Kingsford “CMSC 451: Closest Pair of Points”, https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/closepoints.pdf, Department of Computer Science, University of Maryland

[7] (2017) “Closest pair of points”, https://www.youtube.com/watch?v=6UBDkbVhJckab_channel=DesignandAnalysisofAlgorithms, em: Design and Analysis of Algorithms, YouTube.