Trabalho 1 de Projeto e Análise de Algoritmos
Problema do Par Mais Próximo

Felipe N. Wolter[1], **Luan Carlos Klein**[1]

[1]Engenharia de Computação - DAINF
Universidade Tecnológica Federal do Paraná (UTFPR)
Curitiba – PR – Brasil

Resumo. *Esse relatório é referente ao problema do par mais próximo, analisando a resposta temporal e as diferenças de comportamento entre o algoritmo de força bruta e o algoritmo de divisão e conquista implementados na linguagem C. Foram realizados experimentos para observar o tempo necessário para a obtenção da solução do problema de cada um dos algoritmos em função do número de pontos de entrada. Foi observado que, por ser assintóticamente menor que o algoritmo de força bruta, o método de divisão e conquista apresentou um resultado muito superior para casos onde o tamanho de entrada é grande, a partir de valores da ordem de* 10^3.

1. Algoritmo de Força Bruta

O algoritmo de força bruta consiste em calcular a distância entre todos os pontos da entrada e retornar o menor valor.
Vale ressaltar aqui que os pontos de menor distância são retornados por referência. Para simplificar o pseudocódigo, essa parte trivial foi omitida.

- **Entrada do algoritmo:** V: Vetor de pontos, n: Quantidade de pontos
- **Saídas do algoritmo:** menor distância entre dois pontos em V

Algorithm 1: Força Bruta (V, n)

```
distancia_min = +infinito
while i de 1 até n-1 do
    while j de i+i até n do
        distancia_aux ← calcular_distancia_quadratica(A[i], A[j])
        if distancia_aux < distancia_min then
            distancia_min ← distancia_aux
return raiz_quadrada(distancia_min)
```

O primeiro *while* do pseudocódigo é executado n vezes. Para o segundo *while*, o número de execuções pode ser calculado pela equação:

$$\sum_{k=1}^{n} n - k = \frac{(n^2 - 1)}{2} \tag{1}$$

Cosiderando que o cálculo da raiz quadrada será feito fora do *loop*, a função para cálculo da distância consiste apenas em contas de soma e multiplicação, ou seja, é O(1) e será realizada o mesmo número de vezes que o segundo *while*. O cálculo da raiz quadrada, feito através da biblioteca *math.h*, que usa o método de Newton (utilizando séries) e é O(1), é executado uma vez.
Com isso, a complexidade do algoritmo é definida pela equação 1, logo, é $O(n^2)$.

2. Algoritmo de Divisão e Conquista

Para a construção do algoritmo de divisão e conquista, foi realizada uma pesquisa em busca de ideias de algoritmos pré-concebidos para tal finalidade. A ideia utilizada nesse trabalho foi inspirada em [Indyk 2003], e consiste em ordenar os pontos em relação a x e y, dividir em dois vetores de acordo com a coordenada x do ponto central (med), e encontrar recursivamente a menor distância no lado esquerdo e direito. Após isso, usa-se a menor dessas duas distâncias (d) para realizar apenas as comparações entre pontos que ficaram em lados distintos que estão em uma mesma região de área $2d \times 2d$. Terminado esse processo, retorna a menor distância encontrada e os respectivos pontos.

2.1. Algoritmo

O algoritmo consiste nas seguintes etapas:

- Pré-processamento e ordenação dos pontos, feito na função "divisão e conquista": Nessa etapa, o vetor de entrada é copiado para dois vetores auxiliares, um que será ordenado em relação a x e outro que será ordenado em relação a y. A ordenação é feita pelo algoritmo de *merge sort*. Para evitar erros de execução em casos onde dois pontos apresentam o mesmo valor de x, foi usada a coordenada y como critério de desempate para a ordenação em x.
  A raiz quadrada é calculada uma única vez, antes de retornar o valor final.
- Conquista: Fase que trata do caso base, quando há 3 ou menos pontos para serem comparados. É feito utilizando o algoritmo de força bruta.
- Divisão: Encontra o ponto central do vetor ordenado em relação à coordenada x e divide os pontos em dois grupos: os à sua esquerda e os à sua direita.
  O vetor ordenado em x pode ser separado trivialmente e o vetor em y deve ser percorrido e cada ponto separado individualmente. Em casos especiais onde existem múltiplos pontos com o mesmo valor de x, o critério estabelecido na ordenação garante a separação correta.
- Combinação: usa a menor das distâncias retornadas na divisão (d) para descartar os pontos que não se encontram na região onde $|med - x| \leq d$. Os pontos a esquerda que restarem são comparados com os pontos da direita nos quais $|yd - ye| \leq d$, onde yd se refere ao y do ponto da direita e ye ao do ponto da esquerda, resultando em uma caixa esparsa de dimensões $2d \times d$ (difere da mencionada na 1 pois diz respeito apenas ao lado direito). Feito isso, basta retornar o menor valor encontrado. A ordenação em Y garante que cada ponto da direita seja comparado apenas aos pontos em sua caixa esparsa mais n comparações fora da caixa, pois pode parar de comparar ao achar um ponto acima de sua caixa.
  A figura 1 ilustra a caixa esparsa relativa ao pontos a serem comparados com um ponto P1, delimitada pela linha vermelha central, pela linha verde a direita e pelas linhas amarelas.

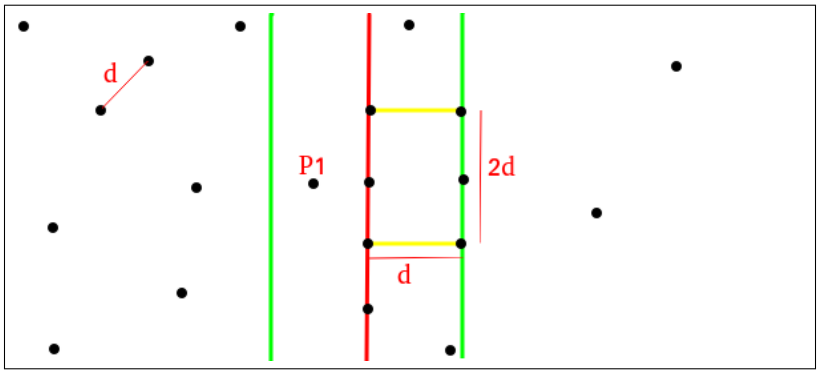


Figura 1. ilustração da caixa esparsa de pontos relativa a um ponto P1

Pra provar a corretude de nosso algoritmo, provaremos ele através do método invariante de laço. A **hipótese inicial** é de que o algoritmo *Find Closest* retorna os dois pontos mais próximos de um vetor ordenado.

O **caso base** se dá quando a quantidade de pontos é inferior a 3. Nesse caso, é chamado o algoritmo de força bruta para esses 3 pontos, retornando assim, a menor distância entre dois pontos.

Para todo vetor de tamanho n$>$ 3, o algoritmo subdivide o vetor em novos dois sub-vetores de tamanho n/2, e chama ele recursivamente. Pela hipótese, essas duas chamadas recursivas retornam a menor distância entre os pontos dentro dele de cada um dos dois vetores. Em seguida, na etapa de combinação, são verificadas as distâncias entre os pontos que estão em lados opostos e suas respectivas distâncias. As únicas comparações ignoradas nessa etapa são entre pontos A e B onde$|A.x - B.x| \geq d$ ou $|A.y - B.y| \geq d$, o que implica, pela desigualdade triangular, que a distância entre A e B é maior que a distância mínima d e assim, não tem influência no resultado . Com isso, a menor distância entre os pontos é encontrada e retornada pelo algoritmo.

Quando a execução retorna para a função "Divisão e Conquista", é sinal de que todas as demais recursões invocadas pela função *Find Closest* também já foram retornadas, e que todo o vetor já foi analisado. Dessa maneira, pelo invariante definido anteriormente, a função retorna a menor distância entre dois pontos do vetor entrada.

Pseudocódigo do pré-processamento e ordenação:

- **Entrada do algoritmo:** V: Vetor de pontos, n: Quantidade de pontos
- **Saídas do algoritmo:** menor distância entre dois pontos em V

Algorithm 2: divisao e conquista (V, n)

```
copy (V, V_x) //Copia o vetor de entrada para dois vetores auxiliares
copy (V, V_y)
merge_sort(V_x) // Ordena o vetor pelo eixo x
merge_sort(V_y) // Ordena o vetor pelo eixo y
distância <- recursivo(V_x, V_y, n) // Chama o algoritmo recursivo
return raiz_quadrada(distância)
```

Pseudocódigo da etapa de divisão e conquista propriamente dita:

- **Entradas**: V_x, V_y: Vetor de pontos ordenados em x e y respectivamente, n: Quantidade de pontos
- **Saída**: menor distância entre dois pontos em V

Algorithm 3: Find Closest

```
//CONQUISTA
if n < 3 then
    return força_bruta(Vx, n)
//DIVISÃO
mid ← (n−1)/2 // Encontra a mediana em x
while i de 1 até n do
    //separa o vetor de y em esquerda e direita
    //para usar nas chamadas recursivas
    if V_y[i].x≤mid then
        ADICIONA(V_y_l, V_y[n])
    else
        ADICIONA(V_y_r, V_y[n])
//chamada recursiva dos pontos a esquerda da mediana
dist_l ← recursivo(V_x_l, V_y_l, size_l)
//chamada recursiva dos pontos a esquerda da mediana
dist_r ← recursivo(V_x_r, V_y_r, size_r)
d ← MIN(dist_r, dist_l)
//COMBINAÇÃO
//vai eliminar os pontos que nao possuem um x dentro da faixa analisada
while i de 1 até n/2 do
    if V_y_l[i].x < mid - d then
        REMOVE(V_y_l, i)
    if V_y_r[i].x > mid + d then
        REMOVE(V_y_r, i)
//min_r evita comparação com pontos que ficaram para baixo
min_r = 1
//para cada ponto na direita, compara com os pontos
//da esquerda dentro de sua na caixa esparsa
while i de 1 até V_y_l.size do
    while j de min_r até V_y_r.size do
        if V_y_l[i].y>V_y_r[j].y+d then
            //ainda não entrou na caixa
            min_r ← j
            continue
        if V_y_l[i].y<V_y_r[j].y+d then
            //já saiu da caixa
            break
        //está na caixa, compara
        if d<DISTÂNCIA(V_y_l[i], V_y_r[j]) then
            d ← DISTÂNCIA(V_y_l[i], V_y_r[j])
return d
```

2.2. Análise de Complexidade de Tempo

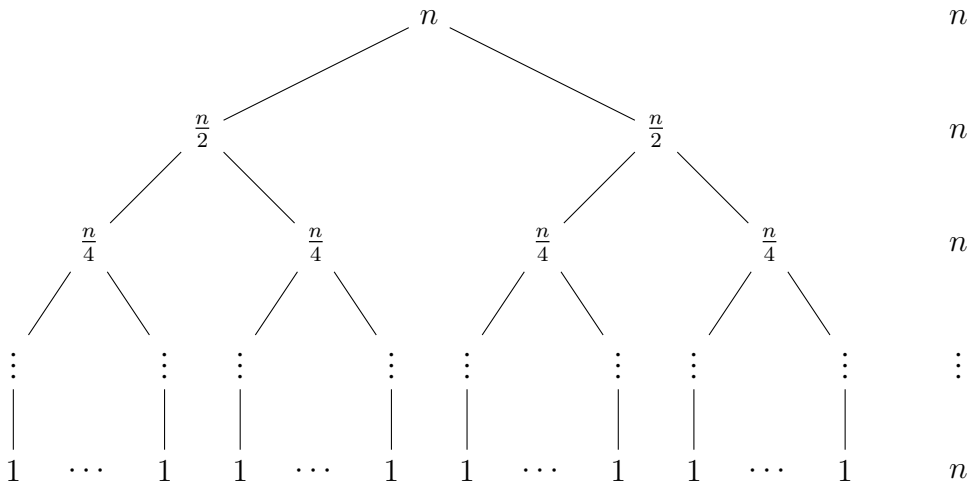
Na primeira etapa, onde ocorre a ordenação dos pontos, o *merge sort* é chamado duas vezes, o que apresenta um custo da ordem de Θ (n$\times$ lg(n)).
No que diz respeito à etapa de divisão e conquista propriamente dita, na função *Find Closest*, ela é separada entre as etapas de conquista, divisão e combinação:

- **Conquista**: uma vez que só é executada entre três pontos ou menos, representa um custo constante ($\Theta(1)$).
- **Divisão**: são feitas duas chamadas recursivas de vetores de tamanho $\frac{n}{2}$, além de percorrer o vetor V_y(de tamanho n). Logo, a divisão é feita em:
  $2 \times T(\frac{n}{2}) + O(n)$.
- **Combinação**: O algoritmo percorre um vetor de tamanho $\frac{n}{2}$ antes de realizar a comparação entre os pontos dentro das caixas esparsas. A combinação pode ser feita em tempo linear, pois em uma caixa de dimensão d $\times$ 2d, na qual a distância mínima entre os pontos é d, a quantidade máxima de pontos que pode haver dentro dela é de 6 pontos [Indyk 2003], como ilustrado na 1. Como cada ponto a esquerda da mediana será comparado a um número constante de pontos, essa etapa tem custo de $O(n)$.

Combinando as etapas da parte de divisão e conquista, é obtida a seguinte relação de recorrência:

$$T(n) = 2 \times T(\frac{n}{2}) + O(n) \tag{2}$$

Dessa relação, é obtida a seguinte árvore de recursão [Thomas H. Cormen 1990]



O caso base será obtido no nível i = lg(n). nesse caso,

$$T(n) = n \times \sum_{i=1}^{lg(n)} i \tag{3}$$

$$T(n) \in O(n \times lg(n)) \tag{4}$$

Para comprovar o comportamento assintótico, será apresentada a prova por substituição. É assumida a hipótese de que:

$$T(m) \leq cm \times lg(m) \tag{5}$$

é válida para para $m < n$. Substituindo na equação de recorrência, é obtido:

$$T(n) = 2T(\frac{n}{2}) + n \tag{6}$$

$$T(n) = 2(c\frac{n}{2} \times lg(\frac{n}{2})) + n \tag{7}$$

$$T(n) = cn \times lg(n) + cn \times lg(\frac{1}{2}) + n \tag{8}$$

$$T(n) = cn \times lg(n) - cn + n \tag{9}$$

$$T(n) \leq cn \times lg(n) \text{ Para } c \geq 1 \tag{10}$$

Como a ordenação e a divisão e conquista ambos possuem complexidade $O(n \times lg(n)$, podemos afirmar que, assintoticamente, o algoritmo como um todo é da ordem de $O(n \times lg(n))$.

3. Experimentos

Para a realização dos experimentos, utilizou-se o algoritmo de geração de pontos disponibilizado pelo professor. Construiu-se um pequeno *script* que executava os teste de maneira automatizada, utilizando sempre os mesmos pontos tanto para o algoritmo de força bruta como para os de divisão e conquista. Visando abranger uma ampla gama de pontos, foram realizados testes variando a quantidade entre 10 a 1 milhão, aumentando de maneira gradativa. Iniciou-se com 10 pontos, com incremento por execução também de 10, e a cada múltiplo inteiro de 10 $\times$ incremento, o incremento era multiplicado por 10.

Os gráficos apresentados na Figura 2 são referentes aos resultados obtidos nos testes. No primeiro gráfico (do lado esquerdo) se apresenta a curva encontrada para a execução com até 10.000 pontos. Já o gráfico do lado esquerda apresenta a curva até 1 milhão de pontos, sendo que esta apresenta o eixo X na escala logarítmica. Observa-se nos gráficos que o algoritmo de força bruta (curva em azul), apresenta um crescimento muito maior que a divisão em conquista (curva em vermelho). Essa diferença se manifesta de maneira bem marcante após 100.000 pontos, onde a força bruta começa a ter um consumo de tempo muito maior que do que a divisão e conquista.

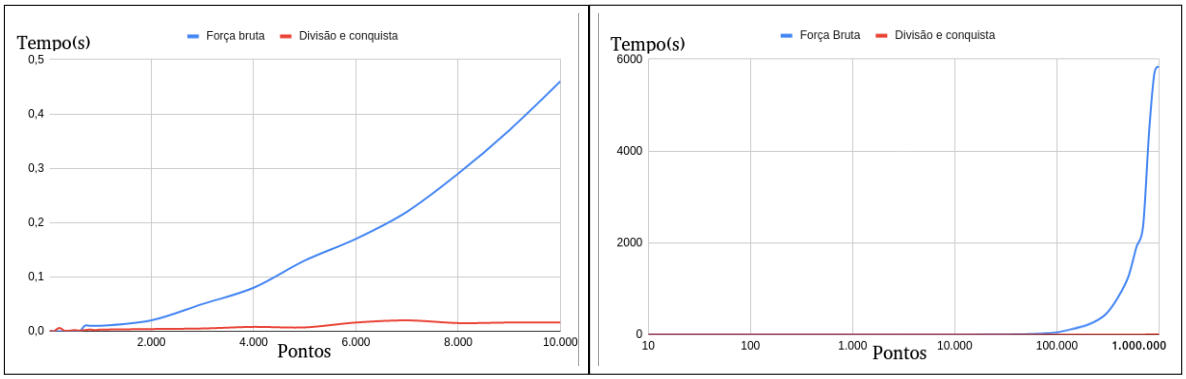


Figura 2. Comparação dos algoritmos de força bruta e divisão e conquista até 10 mil(a) e 1 milhão de pontos(b)

Para fins de comparação numérica, até 1.000 pontos, os tempos de execução de ambos os algoritmos são praticamente idênticos, pois estão muito próximos de 0.00 segundo. Entretanto, ao alcançar a quantidade máxima de pontos testados de 1.000.000, o algoritmo de divisão e conquista levou certa de 2.33 segundos, enquanto que a força bruta levou 5825.33 segundos (1 hora e 37 minutos), uma diferença de 2500 vezes. O ponto chamado n_0, no qual a divisão e conquista supera o algoritmo de força bruta foi encontrado com 110 pontos.

Além disso, foi implementado outro algoritmo de divisão e conquista, que executa o *merge sort* em y a cada iteração, em oposição ao apresentado, que mantêm o vetor ordenado do começo ao fim. Esse algoritmo possui complexidade de $O(n \times (log(n))^2)$. Para a análise, foi realizada a comparação de tempos de execução entre os dois algoritmos de divisão e conquista. Para tal, utilizou-se a mesma metodologia de testes aplicado no teste anterior, utilizando-se exatamente dos mesmos pontos. Na Figura 3, novamente a escala no eixo x é logarítmica, e a curva em azul representa o algoritmo com chamada de ordenação por iteração, enquanto que a curva vermelha corresponde ao método com uma única chamada de ordenação.

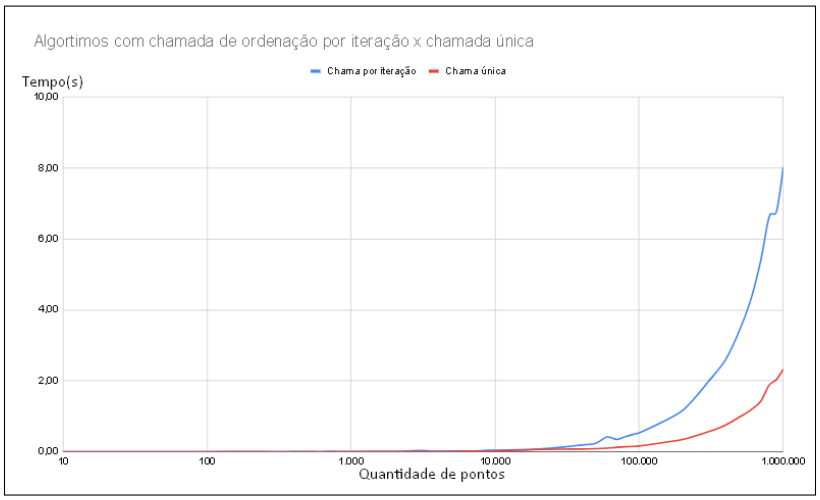


Figura 3. Comparação dos algoritmos de divisão e conquista com chamadas de ordenação por iteração e chamada única utilizando até 1 milhão de pontos

Observa-se novamente que há uma diferença significativa entre as duas curvas, que se evidência a partir de 100.000 pontos. Até 20.000 pontos, ambos as métodos tiveram o mesmo consumo de tempo. A maior diferença se manifesta com 1.000.000 de pontos, onde o algoritmo com chamada única do *merge sort* demora 2.33 segundos, enquanto que o outro leva 8.03 segundos, uma diferença de 245%.

4. Conclusão

Diante dos resultados obtidos, observa-se que quanto mais pontos forem utilizados, maior será a diferença de tempo entre a força-bruta e a divisão e conquista. Uma diferença de 2.500 vezes com 1 milhão de pontos expõe o quão necessário é implementar os algoritmos de maneira eficiente.

Outro ponto relevante é a comparação entre as duas divisões e conquista. Mesmo que a implementação que utiliza múltiplas ordenações seja inferior à outra, ela apresenta uma diferença, ainda que significativa, muito menor que a da força bruta para grandes quantidades de pontos. Esse fato indica que, mesmo com a implementação não ideal da divisão e conquista, ela ainda é muito melhor que a força bruta para o problema estudado, pois apresenta uma complexidade temporal menor.

Por fim, vale destacar que os resultados encontrados foram obtidos em uma máquina específica. Como o processamento entre máquinas difere, os resultados podem ser diferentes em outros computadores, entretanto, o comportamento assintótico apresentado por ambas as curvas permanece.

Referências

Indyk, P. (2003). Closest pair. `http://people.csail.mit.edu/indyk/6.838-old/handouts/lec17.pdf`. Último acesso em 26/03/2021.

Thomas H. Cormen, C. E. L. e. R. L. R. (1990). *Introduction to Algorithms*. Addison-Wesley, 3rd edition.