Universidade Federal De São João Del Rei – (UFSJ)
Departamento De Ciência Da Computação
Curso De Ciência Da Computação

Adélson Oliveira do Carmo Júnior
Renan Rodrigues Sousa

Documentação Trabalho Prático 1 – AEDS III

São João del Rei, MG
2023

1. Introdução

Em um plano, uma quantidade de pontos (x, y) é distribuída, com dois desses pontos sendo as âncoras A = (x_a, 0) e B = (x_b, 0), que formam um segmento horizontal. O objetivo é determinar o maior número possível de pontos que podem ser conectados às âncoras por meio de um segmento de reta, com a condição de que esses segmentos de reta se interceptem **apenas** nas âncoras. A Figura 1 exemplifica as possíveis entradas, a figura mais a direita é objetivo que deve ser buscado.

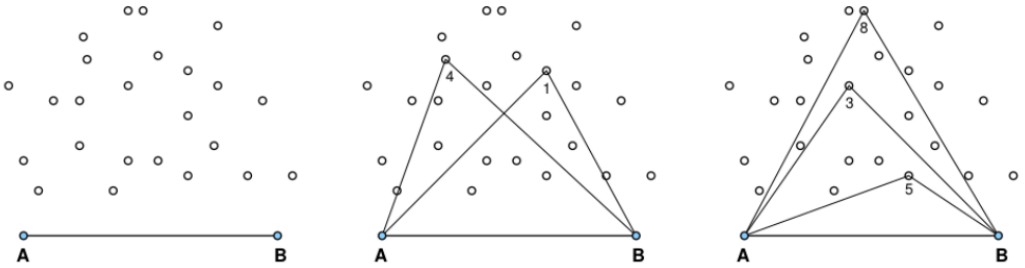

Figura 1. Exemplo do problema e solução

As entradas do problema devem respeitar algumas restrições. O número de entradas não pode ultrapassar 100 pontos. As coordenadas (x, y) devem ser tais que ($0 < x,\ y < 10^4$). As âncoras também devem ter as coordenadas ($0 < x_a < x_b < 10^4$). Não haverá pontos coincidentes e nenhum deles também será colinear as pontas x_a e x_b.

Para executar o código o usuário deve executar o comando: **./main -i <arquivo_de _entrada>.txt -o <arquivo_de_saida>.txt**

Este trabalho propõe encontrar a solução ótima desse problema utilizando Programação Dinâmica e conceitos algébricos de um triângulo.

2. Conceitos Algébricos

O conhecimento de alguns conceitos de álgebra linear é necessário para a solução do problema proposto.

2.1. Área de um triângulo

Para determinar se um ponto está dentro ou não de um triângulo, é necessário utilizar uma função capaz de calcular a área do triângulo. Três pontos podem ser escritos da seguinte forma: $(x_1, y_1), (x_2, y_2), (x_3, y_3)$. Sendo esses os vértices do triângulo, pode-se aplicar uma das propriedades do determinante que afirma que a área de um triângulo pode ser calculada por meio da fórmula: $A = \frac{1}{2} * |D|$. Os vértices do triângulo podem ser representados em formato de matriz, como mostrado na Figura 2 para calcular o determinante.

$$D = \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix}$$

Figura 2. Determinante dos vértices do triângulo

Por conseguinte, pode-se reescrever a fórmula anterior a fim de expressá-la apenas por meio dos vértices. Com essa fórmula podemos determinar qualquer área de um triângulo a partir de seus vértices:

$$Área\ A = \frac{1}{2} * |x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2)|.$$

2.2. Ponto dentro de um triângulo

Utilizando o conceito da área, outra notação matemática é usada para determinar se um ponto está dentro do triângulo. Seja um triângulo ABC e um ponto P, para descobrir se esse ponto está dentro deste triângulo, basta checar se a soma das áreas dos triângulos menores, gerados pelos vértices do triângulo e o ponto (ABP, ACP e BCP), são iguais a área do triângulo ABC. A Figura 3 mostra os triângulos formado pelo ponto dentro do triângulo.

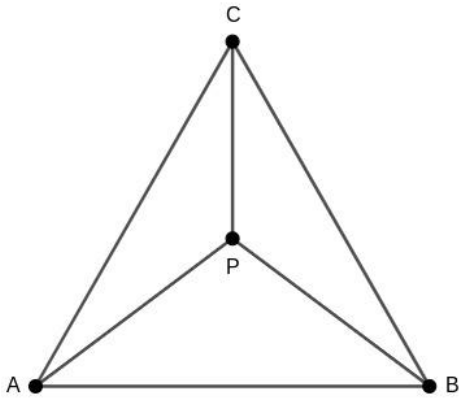

Figura 3. Triângulos internos formados pelo ponto P

Se a soma das áreas menores for igual a área do triângulo ABC, então o ponto está dentro do triângulo. Caso contrário o ponto está fora do triângulo.

3. Programação Dinâmica

Programação dinâmica é um método que busca encontrar a solução geral de um problema, através da análise de uma sequência de subproblemas mais fáceis que o problema original. Os subproblemas menores são processados, os resultados são armazenados e, em seguida, esses resultados são utilizados para resolver o problema geral. Essa abordagem evita que as soluções sejam recalculadas várias vezes, o que otimiza significativamente o tempo de execução do algoritmo.

Há algumas abordagens na programação dinâmica. Na abordagem “bottom-up”, a solução ótima começa a ser calculada a partir do subproblema mais trivial. Essa estratégia é, de certa forma, uma tradução iterativa inteligente de um processo recursivo. Para esse método ser aplicado o problema precisa ter uma estrutura recursiva, pois a solução de cada instância contém a solução de uma subistância.

4. Tipo Abstrato de Dados

O problema abordado neste trabalho foi tratado usando apenas um tipo abstrato de dado. A TAD Ponto possui três atributos, x e y são inteiros correspondentes as coordenadas do ponto. O último atributo, um inteiro chamado cont, armazena o resultado computado para

aquele ponto. Esse valor é o resultado de um subproblema que será usado na programação dinâmica. A estrutura pode ser vista na Figura 4.

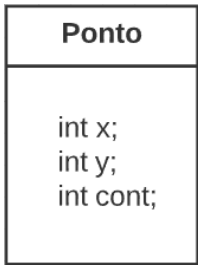

Figura 4. Tipo Abstrato de Dados

5. Listagem de Rotinas

- **Função de alocação do vetor - Ponto *alocar_vetor (int tam):** aloca dinamicamente um vetor de pontos (x, y, cont) na memória.
- **Função de criar vetor - Ponto *criar_vetor (FILE *fp, Ponto *pontos, int n):** essa função lê um arquivo e atribui os valores lidos aos campos x e y de cada ponto. Também inicializa cont igual a zero.
- **Função de criar um Ponto - Ponto criar_ancora (int x): o** objetivo dessa função é criar as âncoras A e B mencionadas na introdução do problema. Cria um ponto definido como (x, 0).
- **QuickSort - troca (Ponto *p1, Ponto *p2) - int particao (Ponto *vetor, int L, int H) - void quickSort (Ponto *vetor, int L, int H):** O algoritmo de QuickSort é composto por três funções. A função "quickSort", recebe o vetor a ser ordenado, bem como os índices de início e fim; a função "particao", que encontra o pivô, divide o vetor em partições e, em seguida, utiliza a função "troca" para rearranjar os pontos no vetor. Esse processo ocorre de forma recursiva até que todo o vetor esteja ordenado.
- **Função de calcular área - float area (Ponto A, Ponto B, Ponto P):** calcula a área de um triângulo usando três pontos, que representam os vértices.
- **Função de contagem – int contagem (Ponto A, Ponto B, Ponto *vetor, int indice):** Função que conta quantos pontos tem dentro de um triângulo formado entre as âncoras e um ponto no plano.
- **Função de análise – bool dentro_triangulo (Ponto A, Ponto B, Ponto C, Ponto P):** Função que analisa se um ponto P está dentro de um triângulo formado por ABC. Retorna um booleano true se está dentro e false se não está dentro.
- **Função máximo – int maximo_pontos (Ponto A, Ponto B, Ponto *vetor, int tam):** Função que testa os pontos e retorna o número máximo de pontos que se conectam somente nas âncoras. Essa função é implementada usando programação dinâmica.

6. Solução do Problema

Inicialmente, devido à estrutura recursiva do problema, foi aplicado um método de força bruta. No entanto, devido à sua alta complexidade computacional e grande consumo de memória, tornou-se inviável. Assim, uma abordagem mais otimizada se fez necessária.

Para solucionar o problema, foi desenvolvido um código em linguagem C que combina os conceitos de álgebra linear já citados com o método de programação dinâmica.

Inicialmente é fornecido um arquivo de entrada que apresenta a seguinte estrutura: na primeira há três números inteiros na forma (n x_a x_b), sendo n o número de pontos, x_a e x_b os valores da coordenada x das âncoras A e B, respectivamente. Nas próximas n linhas estão os valores das coordenadas (x y) de cada ponto. Os dados são lidos, as âncoras são criadas, um vetor de tamanho n é alocado e os pontos armazenados no vetor.

Para o funcionamento da solução é fundamental que o vetor esteja ordenado de forma crescente pela coordenada y. Desta maneira as soluções mais triviais são calculadas primeiro, seguindo o método da programação dinâmica. É utilizado o algoritmo QuickSort para ordenar o vetor e, assim que a ordenação for concluída, aplica-se o método para encontrar a resposta do problema.

6.1. Programação Dinâmica abordagem "bottom-up"

Como já mencionado anteriormente, a abordagem "bottom-up" consiste em calcular as soluções mais simples primeiro. No contexto desse problema específico, isso significa calcular a solução para os pontos mais baixos ou mais perto das âncoras em primeiro lugar.

A solução para este problema está relacionada ao conceito algébrico apresentado na seção 2.2. Para resolver o problema, é necessário conectar um ponto C às âncoras e, em seguida, identificar um ponto P que intercepte C somente nas âncoras. Isso só será possível se P estiver dentro do triângulo formado pelas âncoras e pelo ponto C.

Para realizar esse cálculo é usada a função "maximo_pontos". Essa função recebe as âncoras, o vetor com os pontos a serem testados e o tamanho do vetor. O vetor será percorrido por um loop *for* que vai do primeiro até o último ponto. O primeiro calculo realizado para cada ponto é descobrir quantos outros pontos estão contidos dentro triângulo formado com âncoras. A solução mais trivial é quando não há nenhum outro ponto dentro do triângulo, e neste caso a resposta é 1, somente o próprio ponto. A solução trivial pode ser vista na Figura 5.

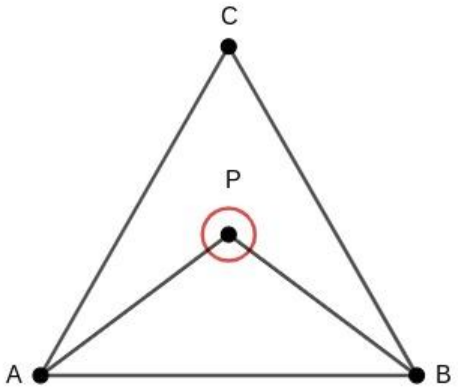

Figura 5. O ponto circulado é uma solução trivial

O resultado obtido para esse ponto é salvo na variável "cont". Dessa forma, não é necessário calcular novamente a solução para esse ponto em problemas futuros. Basta acessar o valor previamente armazenado em "cont".

A segunda possibilidade é que haja um ou mais pontos dentro do triângulo. Neste caso é necessário um segundo loop *for* que percorra o vetor do primeiro ponto até o ponto sendo testado e somente para os pontos que estão dentro triângulo acessar o valor de

“cont” e somar a 1, referente ao próprio ponto. É usado uma variável temporária para armazenar essa soma e comparar com o máximo obtido até o momento, caso seja maior o valor da soma se torna o novo máximo. Um exemplo pode ser visto na **Figura 6**, o ponto com uma “?” está sendo testado e o ponto apontado pela seta possui o maior cont, igual a 2, dentro do triângulo. Nesse exemplo o máximo seria 3.

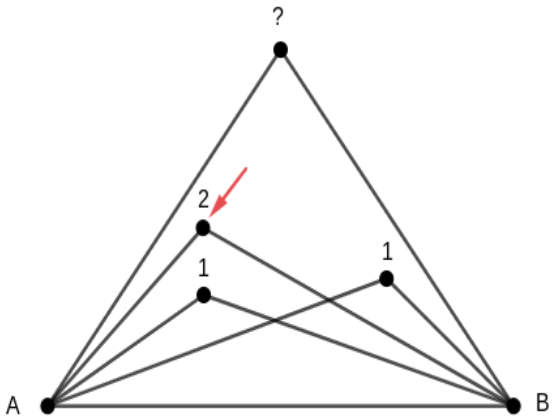

Figura 6. Teste do ponto marcado com “?”

O processo é executado de maneira iterativa até que todos os pontos tenham sido testados. Ao final do processo, a função retorna o maior número possível de pontos que podem ser conectados às âncoras, de modo que haja interseção somente nas próprias âncoras.

7. Análise de Complexidade

Função de alocação do vetor - $\mathbf{O(1)}$: Apenas aloca o vetor que será utilizado para armazenar os pontos.

Função de criar vetor - $\mathbf{O(1)}$: É a função que apenas colocará os elementos lidos dentro do vetor.

QuickSort - $\mathbf{O(n * \log n)}$: Algoritmo de ordenação QuickSort.

Função de calcular área - $\mathbf{O(1)}$: Realiza o cálculo da área entre três pontos.

Função de contagem - $\mathbf{O(i-1)}$: Há *loop for* aninhado que percorre todos os elementos que estão abaixo do ponto analisado. Como sua parada é no zero, a função seria $i-1-0$ ou simplesmente $i-1$.

Função que calcula o máximo de pontos - $\mathbf{O(\max(2n^2, \frac{n^2+3n}{2}))}$: Há dois loops for aninhado, em que o externo vai chamar a função “contador” resultando num somatório de $i-1$ entre 0 e n, tamanho do vetor, como mostrado abaixo:

$$\sum_{i=0}^{n-1}(i-1) = \sum_{i=0}^{n-1}(i) - \sum_{i=0}^{n-1}(1) = \frac{n(n+1)}{2} - n = \frac{n^2+n}{2} - n = \frac{n^2-n}{2}$$

Além disso, o segundo laço de repetição *for* está dentro de um *if*, então apenas no pior caso de cada ponto que ele o fará. Considerando essa possibilidade, deve-se somar

as comparações *if*, já que ele vai ser executado de qualquer maneira, que é o mesmo do tamanho do número de pontos: $n - 0$ ou simplesmente n.

No *for* interno, temos mais três comparações *if*, sendo que esse for vai de 0 até $i - 1$, o que novamente resulta num somatório de $i - 1$ entre 0 e n. Como já fora resolvido anteriormente, sabe-se que o resultado é: $\frac{n^2-n}{2}$ e este deve ser multiplicado por 3, devido aos if, portanto: $3 * \frac{n^2-n}{2}$.

Por fim há mais um *if* que também resultará em n comparações por só estar dentro do for externo. Somando todos os termos no pior caso, obtém-se:

$$\textit{Pior Caso: } \frac{n^2-n}{2} + 3 * \frac{n^2-n}{2} + 2n = 4*\frac{n^2-n}{2} + 2n = 2 * (n^2 - n) + 2n =$$
$$= 2n^2 - 2n + 2n = 2n^2.$$

No melhor caso, o segundo for nunca seria executado, logo seriam apenas:

$$\textit{Melhor caso: } \frac{n^2-n}{2} + n + n = \frac{n^2-n}{2} + 2n = \frac{n^2+3n}{2}.$$

Para sabermos qual função domina assintoticamente, basta analisarmos o resultado de $O(\max(2n^2, \frac{n^2+3n}{2})$, que logicamente é $O(2n^2)$. Sendo assim, $2n^2 \leq c * n^2 \rightarrow 2 \leq c$, logo para qualquer constante maior ou igual a 2, $O(n^2)$ domina assintoticamente ambas as funções.

Ordem de complexidade total - $O(n^2)$: Basta agora ver qual é o max entre cada elemento anterior: $O(max(1,\ 1,\ n * \log n,\ i - 1, 1,\ n^2)) = O(n^2)$.

8. Resultado

Para avaliar o programa, foram utilizadas diversas entradas com diferentes quantidades de pontos. Foi testada uma entrada com 100 pontos, o máximo definido nas restrições do problema, contudo os valores da medição de tempo foram muito baixos. A fim de obter resultados visíveis, foram testadas entradas significativamente maiores do que a da limitação estabelecida na introdução do problema.

Foi possível obter o tempo de usuário e de sistema por meio da função 'getrusage', utilizando as funções contidas no arquivo 'tempo.c'. Com o objetivo de traçar a curva de desempenho do algoritmo, foram realizados testes em seis entradas, todas representando o pior caso em termos de complexidade computacional. As entradas utilizadas continham 100, 250, 500, 750, 1000 e 5000 pontos.

Muito embora o mesmo processo de adicionar pontos fosse testado com o tempo de sistema, ele se mantinha em 0 para todos os valores, sendo inviável fazer um gráfico para mostrar sua variação entre as entradas. Utilizando dos métodos da linguagem R, o gráfico da **Figura 7**, do número de entradas em função do tempo, foi feito.

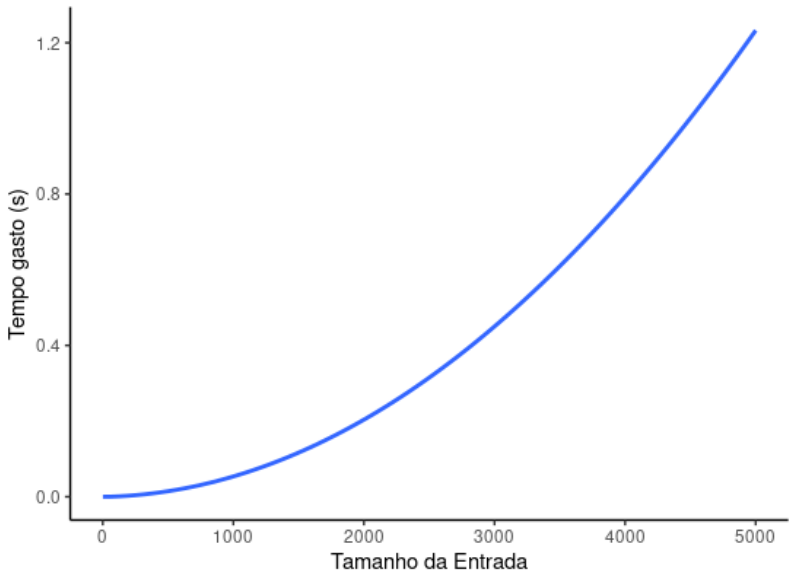

Figura 7. Gráfico tamanho da entrada x tempo para o pior caso

É evidente que a curva do gráfico apresenta um comportamento quadrático, confirmando o comportamento assintótico de $O(n^2)$, proposto na análise de complexidade. Além disso, como mencionado anteriormente, a variação de tempo para a entrada com 100 pontos não é perceptível no gráfico devido ao baixo tempo de execução, apesar de terem sido incluídas nos testes.

Vale a pena comentar que apenas quando foi realizado um teste com uma entrada com cem mil pontos, o tempo de sistema apareceu, sendo igual a 3,99 *10-3 segundos, enquanto o tempo de usuário levou cerca de oito minutos para ser concluído. Isso demonstra o quão inviável era continuar com outros testes.

9. Conclusão

Com os resultados obtidos e com funções de complexidade, foi possível observar uma grande otimização do método da força bruta. Inicialmente o processo recursivo possuía uma grande complexidade (exponencial) e executava processos repetidos, bem semelhante a execução de um Fibonacci recursivo.

Contudo, o método de programação dinâmica foi bem sucedido em otimizar o método da força bruta. Apesar de esse método precisar armazenar o resultado das soluções dos subproblemas, ele ainda é mais eficiente pois o processo anterior necessitava criar vetores auxiliares. Usando programação dinâmica a solução proposta nesse trabalho é capaz de processar o máximo de elementos requisitados de forma eficiente com uma complexidade, no pior caso, de $2n^2$ e é realizado de forma quase instantânea.

No fim, o trabalho demonstrou de forma esclarecedora como a programação dinâmica pode ser utilizada para traduzir um problema com estrutura recursiva em um programa iterativo. Esse resultado confirmou o que foi aprendido em aula, evidenciando que um programa iterativo é consideravelmente mais eficiente do que um programa recursivo.

10. Referências

Gabriel Alessandro de Oliveira (2023) “Área de uma região triangular através do determinante”, Disponível em: https://brasilescola.uol.com.br/matematica/Area-uma-regiao-triangular-atraves-determinante.htm, Abril.

GeeksforGeeks (2023) “Check whether a given point lies inside a triangle or not”, Disponível em: https://www.geeksforgeeks.org/check-whether-a-given-point-lies-inside-a-triangle-or-not/, Abril.

Victor Matheus R. de Carvalho (2019) “Programação Dinâmica”, Disponível em: https://lamfo-unb.github.io/2019/05/30/Programacao-Dinamica/, Abril.