Documentação - Trabalho Prático 1 de Algoritmos e Estruturas de Dados III

Arthur Vieira Silva e Felipe Augusto Moreira Chaves

13 Abril de 2023

Sumário

1 Introdução

Na resolução deste trabalho prático, após analisarmos diversos métodos para conseguir obter a resposta desejada, conseguimos implementar dois algoritmos para tentar solucionar o problema. Inicialmente, foi implementado um **Algoritmo Guloso** no qual, em certos casos, imprimia a solução correta e outros uma solução aproximada e, em seguida, criamos outra solução mais eficiente e que produzia a saída esperada ao final da execução.

1.1 Primeira implementação

A princípio, pensamos em uma maneira de resolver o problema por meio de um **Algoritmo Guloso.** Nessa implementação, após a leitura do arquivo de entrada, os pontos eram ordenados em ordem **decrescente** em relação a coordenada y e, se houvessem coordenadas y iguais, a coordenada x também seria ordenada em ordem decrescente.

Após esse processo de ordenação, iniciaríamos do ponto que tivesse a maior coordenada y e faríamos uma comparação com o segundo ponto da sequência. Se esses dois pontos não se interceptassem fora das âncoras, uma variável máximo seria incrementada e a comparação seria feita entre o ponto que acabou de ser conectado e o próximo ponto da sequência ordenada. Se não, a comparação seguiria normalmente. Após comparar o ponto mais alto com todos os outros pontos, a comparação seria feita do segundo ponto mais alto com os outros $n - 1$ pontos, e assim por diante.

Essa é uma abordagem gulosa pois, considera que um ponto que tenha a coordenada y maior, possivelmente, poderá fazer mais conexões com outros pontos, isto é, escolhe o ponto que parece mais promissor em qualquer instante, e nunca reconsidera essa escolha, independentemente do que venha acontecer no futuro.

Por exemplo, no seguinte conjunto de pontos, a saída mostrada será apenas 3 porém, o correto seria 4 pontos conectados. Isso acontece pois, o algoritmo compara o ponto **C** com o **F** e, em seguida, o **F** com o **D**. Porém, note que apesar do ponto **D** não poder estar conectado ao mesmo tempo que o **F**, com o ponto **C** seria possível. No entanto, essa comparação entre o ponto **C** e **D** já foi descartada.

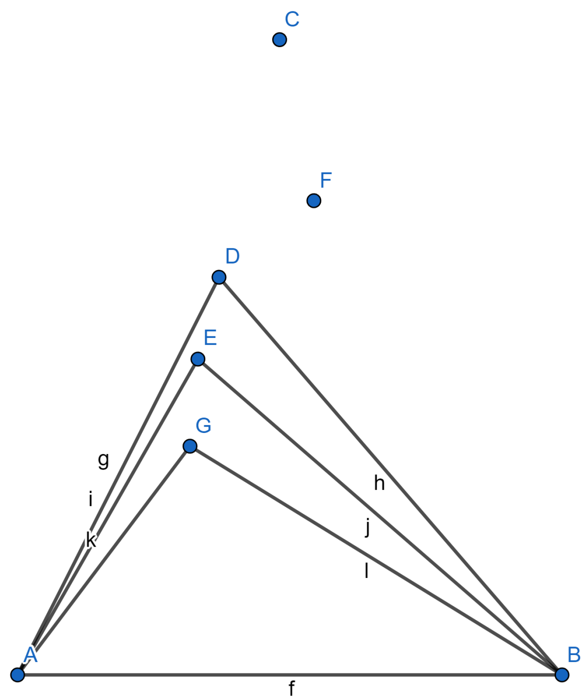


Figura 1: Exemplo gráfico do problema utilizando uma abordagem gulosa

Dessa maneira, a implementação feita por meio de um Algoritmo Guloso funciona para alguns casos específicos em que há um maior número de pontos abaixo de um ponto com uma coordenada y maior. Com isso, essa implementação fornece apenas uma saída **aproximada** para algumas **entradas particulares.**

1.2 Segunda implementação

Após analisarmos o algoritmo então descrito, observamos que para atingir a resposta pretendida, o ideal seria ordenar os pontos em ordem **crescente** em relação a coordenada y e, se houvessem coordenadas y iguais, a coordenada x também seria ordenada em ordem crescente.

Após esse procedimento apresentado, as comparações seriam feitas partindo do ponto mais próximo à reta AB (ponto na posição i) e o seu antecessor na sequência de pontos (ponto na posição j) já ordenados. Se ambos os pontos podem ser conectados ao mesmo tempo então, o número de conexões do ponto na posição i será o número de conexões que o próprio ponto pode realizar, que incialmente é 1, mais o número máximo de conexões entre os pontos que estão abaixo dele. Em seguida, a comparação será feita entre o ponto na posição $i+1$ e os pontos que estão abaixo dele. Esse processo pode ser exemplificado na seguinte sequência de imagens:

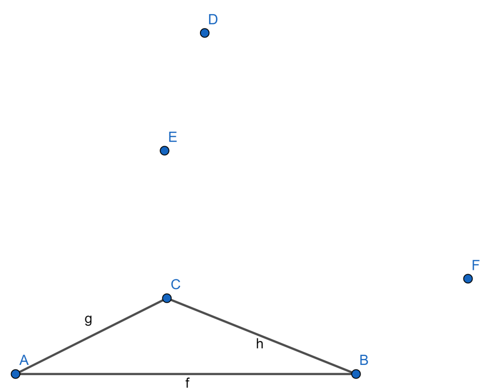


Figura 2: Conexão do ponto C

Note que, como não há nenhum ponto abaixo do ponto **C**, o número máximo de conexões que ele pode fazer é justamente 1. Já o ponto **F** possui o ponto **C** abaixo dele porém, os dois pontos não podem estar conectados ao mesmo tempo logo, o seu número máximo de conexões também é 1.

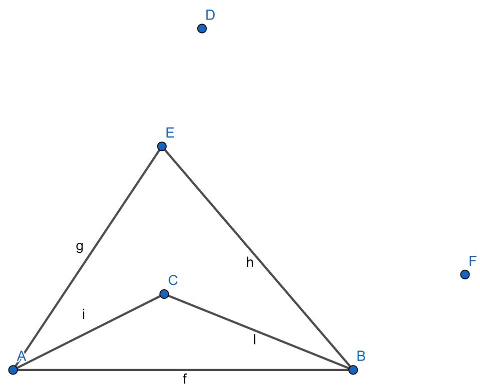


Figura 3: Conexão entre os pontos E e C

Na **Figura 3**, note que abaixo do ponto **E** há os pontos **C** e **F**, no entanto, ele só pode ser conectado ao mesmo tempo com **C**. Dessa maneira, o número de conexões do ponto **E** será o número de conexões dele mesmo acrescido ao número de conexões do ponto **C**, ou seja, 2 conexões.

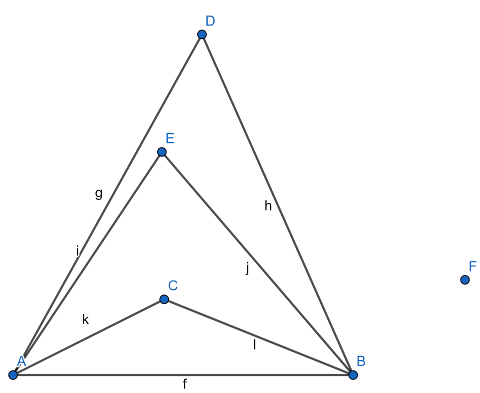


Figura 4: Conexão entre os pontos D, E e C

Por fim, é possível visualizar que na **Figura 4**, os pontos **C**, **E** e **F** estão abaixo do ponto **D** contudo, ele não pode estar conectado ao mesmo tempo que o ponto **F.** Nesse caso, o número de conexões do ponto **D** será o número de conexões do próprio ponto acrescido do maior número de conexões entre os ponto **C** e **E**, que é 2 conexões. Assim, a saída do exemplo dado será 3.

Desse modo, fomos capazes de desenvolver um algoritmo eficiente e que contorne o erro do algoritmo anterior, resolvendo, assim, o problema proposto para diferentes entradas em diferentes situações.

2 Listagem das rotinas

Para desenvolver esse algoritmo, foram utilizadas algumas funções que facilitaram tanto o processo de construção do código quanto o entendimento do programa que estava sendo desenvolvido. A seguir está uma breve descrição dessas funções:

- **LerPontos:** Essa função foi utilizada apenas para ler o arquivo de entrada fornecido. As coordenadas de todos os pontos foram armazenadas em uma *struct* chamada **ponto**.

- **InsertionSort:** Também foi-se utilizado um procedimento para ordenar os n pontos em ordem crescente. Para isso, implementamos o algoritmo de ordenação por inserção. O algoritmo percorre os pontos iniciando com o índice 1 e incrementa esse índice até o último ponto, ordenando cada ponto no *subarray* à esquerda do índice.

- **ProduzirVetores:** Utilizamos esta função para produzir, para cada ponto, dois vetores referentes a um ponto C: $u = AC = C - A$ e $v = BC = C - B$, onde u e v serão armazenados em uma *struct* chamada **vetor**.

- **MaximodePontos:** Esta é a função na qual realemente produz a saída final do programa. Nela usaremos o conceito de produto vetorial para verificar se dois pontos podem estar conectados ao mesmo instante. Será realizado o produto vetorial da seguinte maneira: $u_1 \times u_2 = (x_1 \cdot y_2) - (x_2 \cdot y1)$ e $v_1 \times v_2 = (x_1 \cdot y_2) - (x_2 \cdot y1)$. Onde $u1$ e $v1$ são referentes a um ponto **C** e $u2$ e $v2$ são referentes a um ponto **D**. Se, $u_1 \times u_2 < 0$

e $v_1 \times v_2 > 0$, os pontos podem estar conectados ao mesmo tempo. A figura a seguir mostra essa comparação entre os vetores de um ponto **C** e **D**:

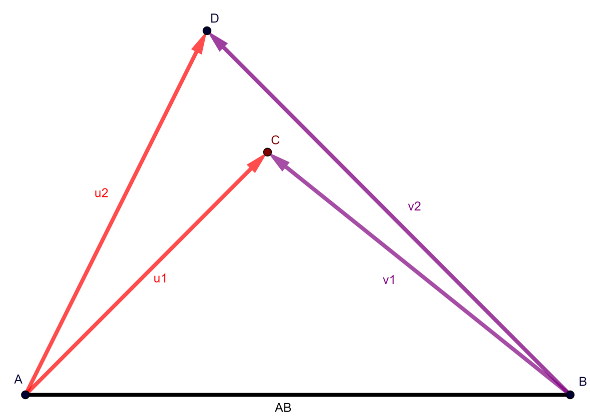


Figura 5: Comparação entre os vetores de dois pontos

Além disso, foi-se utilizado um *array* que armazena o número máximo de conexões feitas por cada ponto, além de uma variável que armazenasse o valor do ponto em uma posição j que possuísse o maior número de conexões, na qual seria ascrescida ao número de conexões de um ponto na posição i. Antes do índice i ser incrementado, ainda há uma comparação que verifica se esse é o ponto que possui mais conexões entre os pontos analisados até então.

No final desta função ela retorna o valor esperado na saída, que é maior número de conexões feitas entre os n pontos do conjunto.

- **SalvarOutput:** Por fim, esta função irá apenas salvar o arquivo de saída imprimindo o valor retornado pela função MaximodePontos.

3 Análise de Complexidade das rotinas

Agora iremos determinar qual a **função de complexidade** e a **ordem** do algoritmo implementado. Para isso, iremos considerar apenas as operações mais significativas realizadas pelo programa, que é o número de **comparações.** Seja uma **função de complexidade f**, em que $f(n)$ é a medida do tempo necessário para executar o algoritmo para um problema de tamanho n.

Vamos definir a função de complexidade e ordem de cada rotina individualmente e, por fim, de todo o código.

Na função de leitura do arquivo de entrada, temos algumas comparações que são realizadas apenas uma vez, isto é, $O(1)$ e temos um laço que será executado n vezes, de maneira uniforme sobre todas as entradas de tamanho n. Logo, $f(n) = n$, para $n > 0$, para o melhor caso, pior caso e caso médio.

Além disso, temos um algoritmo de ordenação **Insertion Sort** no qual, o número mínimo de comparações ocorre quando os itens já estão ordenados, e o número máximo quando os itens estão na ordem reversa. Em seu anel mais interno, na i-ésima iteração, o valor de C_i será:

- Melhor caso: $C_i = 1$;
- Pior caso: $C_i = i$;
- Caso médio: $C_i = \frac{1}{i}(1 + 2 + ... + i) = \frac{i+1}{2}$.

Onde C é o número de comparações entre os elementos. Considerando que todas as permutações de n são igualmente prováveis no caso médio, temos que:

- Melhor caso: $C(n) = (1 + 1 + ... + 1) = n - 1$;
- Pior caso: $C(n) = (1 + 2 + ... + n - 1) = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}$;
- Caso médio: $C(n) = \frac{1}{2}(3 + 4 + ... + n + 1) = \frac{n^2}{4} + \frac{3n}{4} - 1$.

A análise da função para produzir os vetores é exatamente a mesma da primeira função, ou seja, o laço será executado n vezes, de maneira também uniforme sobre todas as entradas de tamanho n. Logo, $f(n) = n$, para $n > 0$, para o melhor caso, pior caso e caso médio.

Para concluir, na função MaximodePontos temos dois *loops*, um externo e outro mais interno, em que, serão executados de maneira uniforme sobre todas as entradas de tamanho n, temos que: $f(n) = (1 + 2 + ... + n - 1) = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}$, para o melhor caso, pior caso e caso médio.

Portanto, somando as funções de complexidade de cada procedimento:

- Melhor caso: $f(n) = n + 1 + n + (\frac{n^2}{2} + \frac{n}{2}) = \frac{n^2}{2} + 2n + 1$;
- Pior caso: $f(n) = n + 2(\frac{n^2}{2} + \frac{n}{2}) + n = n^2 + 3n$;
- Caso médio: $f(n) = n + (\frac{n^2}{4} + \frac{3n}{4} - 1) + n + (\frac{n^2}{2} + \frac{n}{2}) = \frac{7n^2}{4} + \frac{9n}{4} - 1$.

Assim sendo, o tempo de execução do programa é $O(n^2)$, para o melhor caso, pior caso e caso médio, isto é, trata-se de um algoritmo de **complexidade quadrática.** Desse modo, este um algoritmo útil para resolver o problema proposto, tendo em vista que a entrada não será maior que 100 pontos, o que é um tamanho relativamente pequeno.

4 Análise dos resultados obtidos

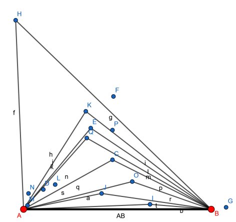
Após vários testes, selecionamos três exemplos que consideramos mais relevantes para as nossas análises tanto do tempo de sistema quanto do tempo de usuário, e sua relação com os tempos do relógio. Para isso, utilizamos as funções **getrusage** e **gettimeofday.**

O tempo de usuário representa o tempo de CPU gasto no código no modo usuário (fora do *kernel*) dentro do processo, é apenas o tempo real de CPU usado durante a execução do programa. O tempo de sistema é a quantidade de tempo da CPU gasto durante o período que o programa executa no modo *kernel*. Já o tempo de relógio é todo o tempo decorrido, isto é, do início ao fim da chamada, inclui o tempo utilizado em outros processos e o tempo que a execução fica parada esperando terminar operações de entrada/saída.

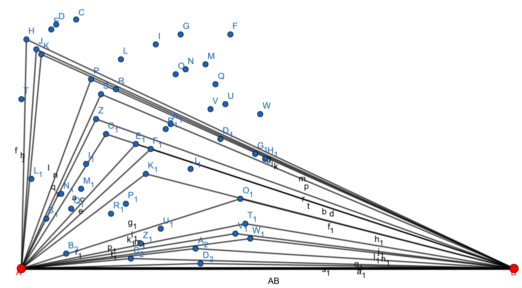
A seguir está um exemplo gráfico dos três exemplos citados anteriormente:

Figura 6: Exemplos testados para análise dos resultados

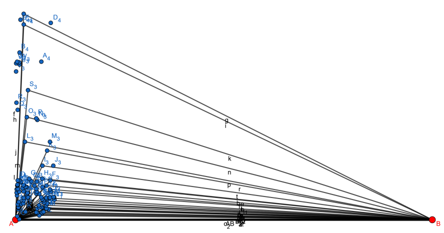
(a) Teste com 15 pontos



(b) Teste com 50 pontos



(c) Teste com 100 pontos



O primeiro teste foi realizado com 15 pontos dados em ordem aleatória e a saída obtida foi 8. Note que como não há um valor grande de pontos, o tempo necessário para execução do programa não foi muito alto. O tempo de usuário variou de $0,000379$ segundos até $0,001414$ segundos aproximadamente, o tempo de sistema foi, na grande maioria dos casos, igual a 0 mas, em algumas execuções, foi um valor próximo a $0,000426$ segundos. Já o tempo de relógio, obtido a partir da função **gettimeofday**, variou entre $0,000374$ segundos e $0,001413$ segundos.

No segundo teste, o valor de pontos na entrada foi 50 pontos ordenados em ordem decrescente, o que acarreta no pior caso do algoritmo de ordenação por inserção, e o valor da saída foi 17 pontos conectados. Neste caso, os valores obtidos nos tempos podem ser um pouco mais altos, devido a maneira como os pontos foram lidos no arquivo de entrada. O tempo de usuário variou de $0,000836$ segundos a $0,001474$ segundos. O tempo de sistema oscliou entre $0,000657$ segundos e $0,001912$ segundos em apenas algumas das execuções testadas. Já o tempo de relógio variou entre $0,000830$ segundos e $0,001925$

segundos.

Por fim, o último teste foi feito com 100 pontos já ordenados em ordem crescente e o valor obtido na saída foi de 27 pontos conectados. O tempo de usuário teve uma variação de $0,000939$ segundos até $0,002367$ segundos. O tempo de sistema também foi igual a 0 em várias execuções, mas em algumas obtemos um resultado entre $0,000697$ segundos e $0,001440$ segundos. Já o tempo de relógio variou de $0,000936$ segundos a $0,002364$ segundos. Portanto, note que apesar dos pontos já estarem ordenados, como o número de conexões foi maior do que no segundo teste, possivelmente, por isso o tempo de execução também foi mais elevado.

5 Referências

ZIVIANI, N. **Projeto de Algoritmos: com implementações em PASCAL e C.** 3 ed. São Paulo: Cengage Learning, 2013.

Steimbruch, A; WINTERLE, P. **Geometria Analítica.** 1 ed. São Paulo: Pearson Makron Books, 1995.