Algorítimos e Estrutura de Dados III
Primeiro Trabalho Prático - Hipercampos

Pablo Cecilio Oliveira
Alexander Cristian

1 Introdução

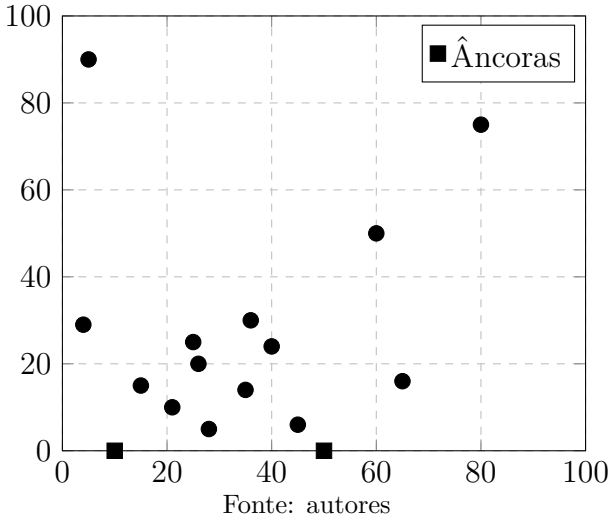
Na Ciência da Computação, o estudo de algorítimos para resolução de problemas geométricos é conhecido como Geometria Computacional. De forma geral, o objetivo deste ramo é resolver de maneira eficiente utilizando o menor número possível de operações sobre os elementos geométricos elementares.[5]

Dentre os problemas geométricos, temos um conhecido "como Hipercampos", o qual pode ser visto como desafio em maratonas de programação[2]. Neste trabalho, é apresentado a solução para esse problema por meio de um algorítimo contido em um programa desenvolvido na linguagem em C.

1.1 Hipercampos, especificação do problema

No problema de Hipercampos, um plano cartesiano em $\mathbb{R}^2$ possui duas "âncoras", dois pontos A e B, onde o eixo Y das duas âncoras são iguais a zero, ou seja $A = (X_A, 0)$ e $B = (X_B, 0)$. Os valores do eixo X das âncoras variam de X_A até X_B, formando assim um segmento de reta horizontal, tal que $0 < X_A < X_B \leqslant 10^4$. (Figura 1)

Figura 1: Exemplo de entrada para o problema.

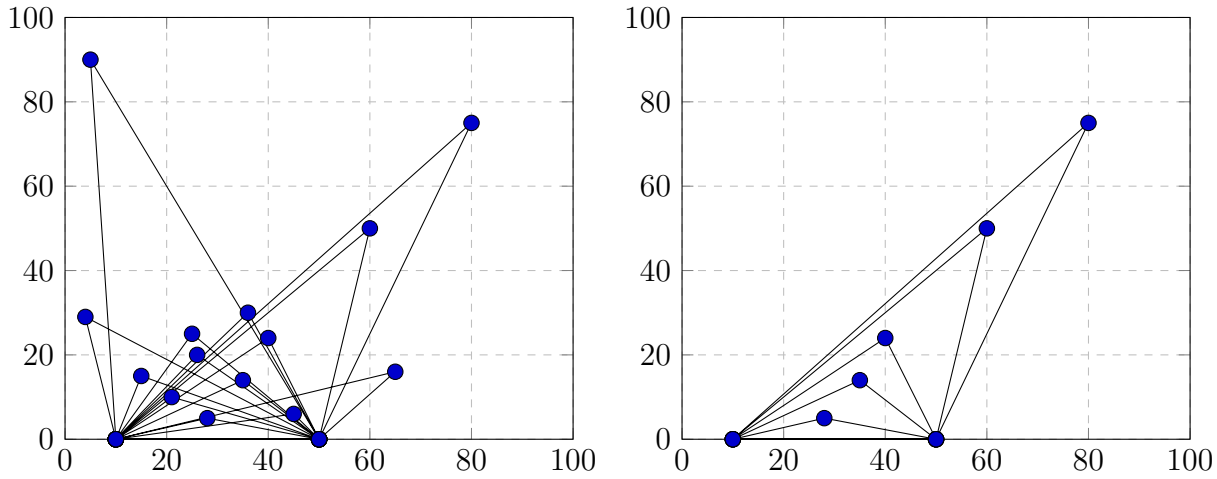


Fonte: autores

Ao plano cartesiano também somam-se um conjunto P de N pontos (X_i, Y_i), sendo que $N(1 \leqslant N \leqslant 100)$. Os pontos do conjunto P podem ter suas coordenadas variando entre 0 até 10^4, ou seja, $0 < X_i, Y_i \leqslant 10^4$.

O objetivo do problema de Hipercampos é ligar os pontos contidos em P às âncoras, formando assim um número máximo de triângulos sem que esses se interceptem (Figura 2). E para esse proposito é apresentado um algorítimo contido no programa apresentado neste trabalho.

Figura 2: Hipercampos, solucionando.



Fonte: autores

1.2 Visão geral sobre o funcionamento do programa

A primeira linha da entrada contém três inteiros, $N(1 \leqslant N \leqslant 100)$, X_A e X_B $(0 < X_A < X_B \leqslant 10000)$ representando, respectivamente, o número de pontos no conjunto P e as abscissas das âncoras A e B. As N linhas seguintes contêm, cada uma, dois inteiros Xi e Yi $(0 < X_i, Y_i \leqslant 10000)$,representando as coordenadas dos pontos, para $1 \leqslant i \leqslant N$. Não há pontos coincidentes e não há dois pontos u e v distintos tais que A, u, v ou B, u, v sejam colineares.

O programa imprime uma linha contendo um inteiro, representando o número máximo de pontos de PP que podem ser ligados com interseção de segmentos apenas nas âncoras.

O método em questão usado para resolver esse problema se baseia na exploração do sistema de coordenadas baricêntricas e na orientação dos segmentos de retas formados pela conexão dos pontos. Portanto em primeiro lugar é verificada a orientação das retas para isolar os casos em que elas se interceptam ou são colineares.

Logo em seguida precisa-se saber quando um ponto esta contido em um triangulo. Uma solução simples seria traçar uma reta que segue horizontalmente para a direita, e depois fazendo comparações para saber quantas vezes ela intercepta o polígono formado, se o resultado for um número par o ponto está fora, se for impar ele está dentro. Porem isso levaria o programa a executar muitas operações, então evoluindo desse conceito chegamos

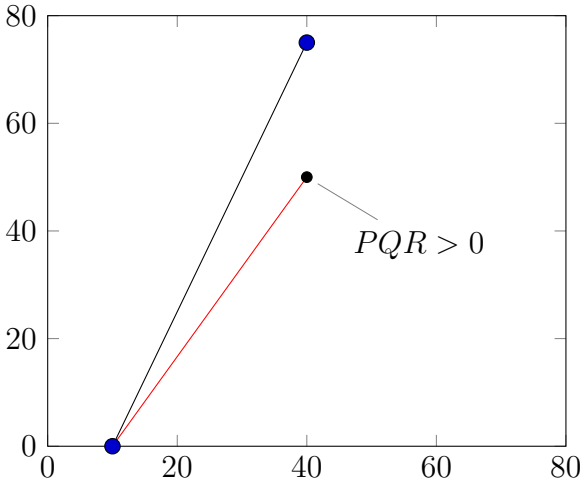
a uma solução usando as coordenadas baricêntricas, onde é verificado em qual lado do meio plano criado pelas arestas está o ponto.

Verifica por meio da expressão de orientação da reta:

$$(y2 - y1)(x3 - x2) - (y3 - y2)(x2 - x1)$$

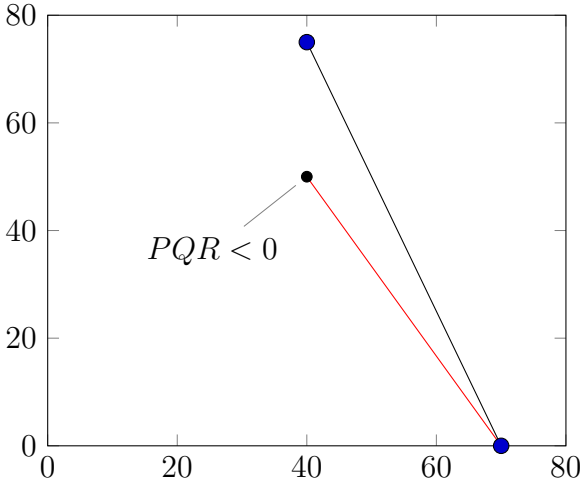
Se os pontos são colineares, e a orientação do triangulo (horário ou anti-horário)

Figura 4a: Determinando o ponto interno ao triângulo



Fonte: autores

Figura 4b: Determinando o ponto interno ao triângulo



Fonte: autores

2 Implementação

Tabela 1: Funções do programa

Funções	Finalidade	Complexidade*
$debug()$	Função que verifica a condição para retorno de possíveis bugs no programa.	$O(1)$
$create()$	Inicializa a Lista encadeada.	$O(1)$
$insere()$	Insere os dados em uma lista encadeada.	$O(1)$
$printCJT()$	Imprime uma Lista encadeada.	$O(n)$
$sizeCJT()$	Retorna o tamanho da lista encadeada.	$O(n)$
$dump()$	Libera a memoria alocada pela lista.	$O(n)$
$isEmpty()$	Verifica se uma lista encadeada está vazia.	$O(1)$
$openFILE()$	Abre o arquivo solicitado e transfere os dados para uma lista encadeada.	$O(n)$
$saveFILE()$	Salva a solução do problema em um arquivo.	$O(1)$
$chkFILE()$	Verifica por possíveis erros de entrada em um arquivo.	$O(1)$
$showerro()$	Retorna possíveis erros no arquivo de entrada.	$O(1)$
$ask()$	Solicita a confirmação do usuário caso erros de entrada sejam encontrados.	$O(1)$
$cpyCJT()$	Copia os dados de uma lista encadeada para outra lista encadeada.	$O(1)$
$PQR()$	Algorítimo de orientação do ponto em relação a reta da ancora.	$O(1)$
$findMAX()$	Função recursiva que determina o maior conjunto de pontos que se encontram dentro do triângulo formado pelas ancoras e um ponto (x, y).	$O(n)$
$soluciona()$ $solucao()$	Funções de chamada e retorno para a execução do algorítimo	$O(1)$
$plotGraph()$	PIPE para o gnuplot com a finalidade de renderizar os arquivos .svg contendo respectivamente, a entrada e saída da solução do problema.	$O(n)$

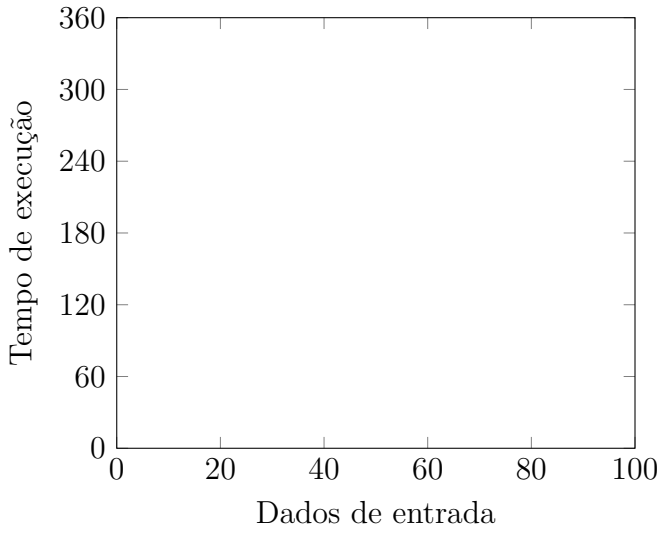
Fonte: autores

3 Análise de Complexidade

$$f(n) = 11 + (n+1) * \frac{n}{n-1} + \frac{n}{2} * \frac{1}{2n} + n + \frac{n}{2} + 4n$$
$$f(n) = 11 + \frac{n^2+n}{n-1} + \frac{n}{4n} + \frac{n}{2} + 5n$$
$$f(n) = \frac{26n^2 + 27n - 45}{4(n-1)}$$

$$T(n) = aT(\frac{n}{b}) + f(n)$$

Figura 5: Tempo de execução do algoritimo



Fonte: autores

4 Considerações finais

O Trabalho computacional 1 da disciplina foi uma grande oportunidade para aprender sobre grafos e LCS, que rodeiam o algoritmo ótimo para a solução desse problema, o que é a introdução para programação dinâmica e acreditamos ser o intuito desse trabalho, também proporcionou um contato maior com a analise de complexidade do algoritmo.

Um dos maiores problemas no desenvolvimento foi encontrar um algoritmo que possuísse um comportamento adequado quando a entrada de valores é muito grande. Apesar da forte base matemática de nossos métodos, em alguns casos eles podem levar a uma falta de precisão, porque o sistema de números de ponto flutuante tem tamanho limitado e na maioria das vezes lida com aproximações. O problema ocorre às vezes quando um ponto p deve estar exatamente na borda de um triângulo, as aproximações levam a falhar no teste.

Para a construção de gráficos que auxiliam em uma melhor visualização do trabalho foi necessário o gnuplot.

Referências

[1] et al. Elin, Kisielewicz. How to determine if a point is in a 2d triangle? https://stackoverflow.com/questions/2049582/how-to-determine-if-a-point-is-in-a-2d-triangle. [Acesso em: 23-Agosto-2018].

[2] URI Online Judge. Hipercampo. https://www.urionlinejudge.com.br/judge/en/problems/view/2665. [Acesso em: 3-Setembro-2018].

[3] Cédric Jules. Accurate point in triangle test. http://totologic.blogspot.com/2014/01/accurate-point-in-triangle-test.html. [Acesso em: 23-Agosto-2018].

[4] Patrick Prosser. Geometric algorithms. http://www.dcs.gla.ac.uk/~pat/52233/slides/Geometry1x1.pdf. [Acesso em: 23-Agosto-2018].

[5] Wikipedia contributors. Computational geometry — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Computational_geometry&oldid=841504892, 2018. [Acesso em: 3-Setembro-2018].