Universidade Federal de São João del Rei
Departamento de Ciência da Computação
Curso de Ciência da Computação

Trabalho Prático 1: uma resolução para o problema de Hipercampos em tempo $O(n^2)$

Rodrigo José Zonzin Esteves

Introdução e abordagem matemática

Breve descrição do problema

Dados dois pontos-âncoras $A = (x_a, 0)$ e $B = (x_b, 0)$ e um conjunto de pontos $\mathbb{P}$, qual a maior sequência possível de segmentos tais que os segmentos $\overline{Ap}$ e $\overline{pB}$ $\forall p \in \mathbb{P}$ se interceptem apenas nos pontos-âncoras.

Pela definição do problema, garantimos as seguintes restrições: $0 < p_x$ *e* $0 < p_y < 10^4$ $\forall p \in \mathbb{P}$ *e* $0 < X_a < X_b < 10^4$.

Natureza geométrica do problema

Determinar se dois segmentos de reta se interceptam é um problema clássico de geometria computacional (Cormen et al., 2009). Um algoritmo que implementa uma técnica para resolvê-lo é conhecido como *Varredura*.

Nossa abordagem, no entanto, se baseia em conclusões geométricas a respeito da avaliação de um triângulo determinado pelos pontos-âncora e um ponto genérico $p \in \mathbb{P}$.

A imagem a seguir, mostra os pontos de um conjunto $\mathbb{P}_1 = \{C(3,4), D(3,2), E(7,4)\}$. Para determinarmos se há intersecção entre os pontos C e E, olhamos para o os triângulos determinados pelos pontos ACB e AEB. A intersecção acontecerá se o ponto E não estiver contido no triângulo ACB **ou** se o ponto C não estiver contido no triângulo AEB.

Se fixarmos o ponto D e compararmo-lo aos pontos C e E, temos que $D \subset ACB$ e $D \not\subset AEB$.

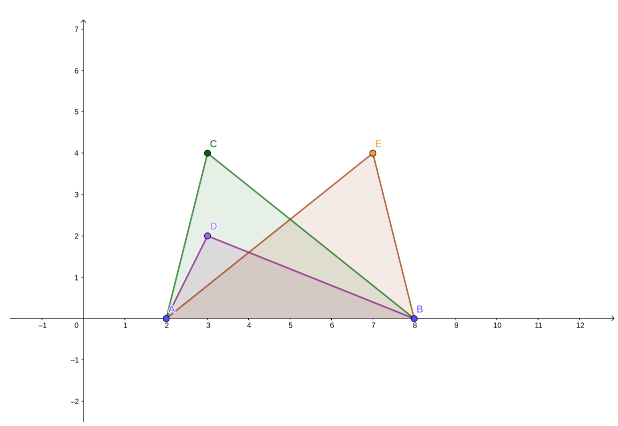


Figura 1: Conjuntos de pontos $\mathbb{P}_1$

Perceba, no entanto, que contar quantos pontos um triângulo ΔABp_i contém e tomar o máximo dentre todas as possibilidades não é suficiente, uma vez que dois pontos p_1 e p_2 podem não formar uma sequência entre si. A imagem a seguir ilustra essa situação.

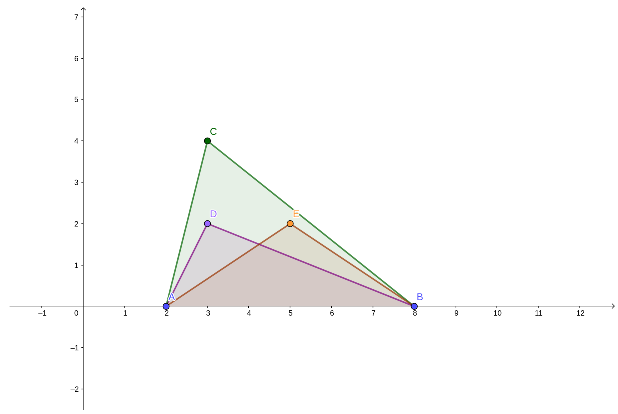


Figura 2: Conjunto de pontos que não satisfazem a proposta de abordagem

Os pontos D e E não formam uma sequência válida entre si. Se contássemos quantos pontos ΔABC contém, encontraríamos 2 como resposta. No entanto, há apenas um arranjo que respeita nossa restrição.

Para solucionar esse problema, adotamos a seguinte abordagem: para um ponto genérico p_i, contamos quantos pontos subjacentes (*i.e.*, contidos em ΔABp_i) ele contém. Se ele não contém nenhum, adicionamos uma flag em p_i que passa a valer 1 (uma sequência válida). Por outro lado, se ele contém pelo menos um ponto abaixo de si, analisamo-o com todos os pontos anteriores $p_j tal\ que\ 0 < j < i$. Se p_j está dentro de p_i, setamos uma variável auxiliar que vale $1 + p_j.tag$ (todas as sequências válidas mais essa). Se a variável auxiliar for maior do que a tag do ponto p_i, então atualizamo-a com o valor da variável auxiliar.

Por fim, analisamos para o ponto p_i se sua tag é maior do que o a sequência máxima até aquele momento. Caso seja, a sequência máxima passa ser a tag do ponto p_i.

O pseudo-código abaixo sintetiza essa abordagem:

```
maior  = 0
  for p_i in ℙ:
    dentro = contar_pontos_subjacentes(ΔABp_i)
    if dentro == 0:
      p_i->tag = 1
    else:
      for p_j ∈ ℙ talque 0<j<i:
        if esta_dentro(p_i, p_j) == true:
          aux = 1 + p_j->tag
        if aux > p_i->tag:
          p_i->tag = aux
    if p_i->tag > maior:
      maior = p_i->tag
return maior
```

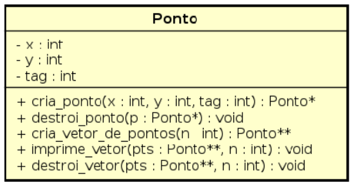
Através dessa abordagem, asseguramos que somente sequências de pontos válidos serão contabilizados durante a análise de um ponto p_i. Analisamos os n pontos disponíveis e salvamos a maior sequência a medida que a formos encontrando.

Estruturas de dados e especificação do código

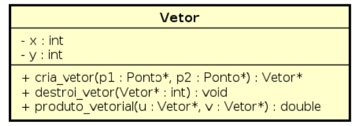
Para modelarmos o problema, utilizamos os Tipos Abstratos de Dados apresentados na figura 3.

O TAD Ponto é o principal tipo utilizado no programa. Essa acepção não só é verdadeira, como elegante, já que dos postulados euclidianos, sabemos:

1. *"Fique postulado traçar uma reta a partir de todo ponto até todo ponto"*. Euclides
   De forma geral, "dois pontos distintos determinam uma reta".



(a) TAD Ponto



(b) TAD Vetor

Figura 3: TADs utilizados

2. *"Figuras retilíneas são as contidas por retas, por um lado, triláteras, as por três [...]"*. Euclides. Dessa definição e de outros postulados segue a famosa expressão: *"três pontos não todos colineares determinam um triângulo"*.

Outra estrutura de dados importante é o tipo Vetor. De acordo com Boulos (2010), a cada ponto $P \in E^3$ e a cada vetor $\vec{v} \in V^3$, a soma de P com $\vec{v}$ é definida em termos de um único representante de $\vec{v} \in V^3$: o segmento orientado (P, Q). Segue:

$$P + \overline{PQ} = Q$$
$$P + \vec{v} = Q$$

Somando $-P$ dos dois lados, temos:

$$\vec{v} = Q - P$$

Se sabemos as componentes x e y de Q e P em uma base qualquer de E^3, podemos definir o vetor apenas subtraindo essas componentes. Nosso programa usa essa abordagem para instanciar um TAD $Vetor$.

Usamos esse resultado pois sabemos da álgebra linear que dado dois vetores $\vec{u}, \vec{v} \in V^3$, o produto vetorial euclidiano $\vec{u} \times \vec{v}$ determina a área do paralelogramo contido entre eles. Anton and Rorres (2012). Calcular área de um triângulo é algo primordial para uma de nossas funções.

Para determinamos se um ponto p_i está dentro de um triângulo ΔABP, verificamos se $|\Delta ABP|$[1] $= |\Delta APp_i| + |\Delta ABp_i| + |\Delta PBp_i|$.

Com o resultado anterior, basta fazer o produto vetorial dos três vetores e verificar o valor se suas somas é igual ao produto vetorial do triângulo mais externo. Note que: $|\Delta ABP| = \vec{AP} \times \vec{BP}$

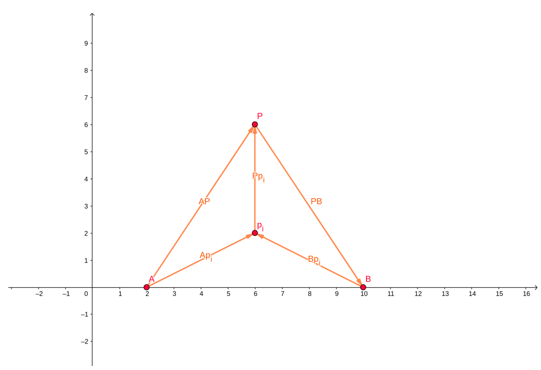


Figura 4: Produtos vetoriais

$|\Delta ABp_i| = \vec{Ap_i} \times \vec{Pp_i}$
$|\Delta APp_i| = \vec{AP} \times \vec{Ap_i}$
$|\Delta PBp_i| = \vec{Bp_i} \times \vec{Pp_i}$

[1] Notação: $|\Delta A|$ = área do triângulo A

Se $|\Delta ABP|$ for igual à soma dos demais vetoriais, então sabemos que o ponto p_i está contido em ΔABP.

Todas as metodologias analíticas foram descritas em módulo separado, para facilitar a interação com a *main*. Além de funções de análise, o módulo analítico também conta com um algoritmo de ordenação para o *array* de pontos. O algoritmo escolhido foi o *Selection Sort*. Optamos por esse algoritmo pois consideramos que o desempenho assintótico $O(n^2)$ é compensado pela facilidade de implementação.

Complexidade e análise dos resultados obtidos

A tabela a seguir lista a ordem de complexidade das principais funções do programa. Definimos a operação comparação como a variável de interesse para a análise de complexidade. Ressaltamos que funções tradicionalmente $O(1)$, como alocar uma estrutura de dados e inicializá-la (construtor), não foram listadas. **As justificativas podem ser conferidas nos comentários do código**.

Função	Complexidade	Justificativa
destroi_vetor_de_pontos()	$O(n)$	É preciso iterar sobre cada ponto e desalocar cada ponto individualmente.
imprime_vetor_de_pontos()	$O(n)$	Iterar sobre cada ponto do array e printa as componentes x e y
cria_vetor()	$O(1)$	Aloca-se o tamanho de um vetor e realiza aritmética de floats para os componentes x e y.
produto_interno()	$O(1)$	Realiza aritmética.
area_triangulo()	$O(1)$	Realiza aritmética e 1 comparação por chamada.
contar_pontos_subjacentes()	$O(n)$	O loop for itera sobre $n-2$ pontos. Portanto, a complexidade do loop é O(n). Dentro do loop, há operações que têm complexidade constantes.
esta_dentro()	$O(1)$	Chama a função cria_vetor() e realiza 2 comparações.
ordena_vetor()	$O(n^2)$	Complexidade do a´lgoritmo Selection Sort. Cormen et al. (2009)
analisa_todos_os_pontos()	$O(n^2)$	No corpo do loop mais externo há a chamada da função contar_pontos_subjacentes(), de complexidade $O(n)$ e de um for interno. Pela propriedade aditiva podemos considerar o corpo do loop externo como um bloco de código de complexidade $O(O_{func}(n) + O_{for\_interno}(n)) = O(n)$. Como cada passagem do for tem complexidade $O(n)$, usamos a propriedade $O(O(n) \cdot O(n)) = O(n^2)$ e obtemos a complexidade final da função.

Tempo de execução *vs* número de entradas

Para verificar o comportamento quadrático do nosso código, simulamos várias execuções para diferentes tamanhos de n.

Primeiramente, definimos aleatoriamente 10^4 pares ordenados usando a linguagem $JavaScript$.

Alteramos nossa *main* e colocamos um for para que ela variasse o range de acesso do vetor de pontos para a função $analisa\_todos\_os\_pontos()$.

Inicialmente, nossa proposta era analisar o tempo de execução para cada execução de $n = 1$ até $n = 10^4$. No entanto, o aumento progressivo do tempo nos fez desistir dessa abordagem.

De fato, se somarmos o tempo de execução estimado por $\hat{f}(n)$ - *definido em seguida* - das 10000 primeiras execuções, encontramos $\int_0^{10k} \hat{f}(n)\ dn = 8h10min$.

Para contornar essa demora, optamos por iterar de 50 em 50, formando um *dataset* de 200 elementos de $n = 1, n = 51, ..., n = 9951$ entradas. Usamos o software R para plotar o gráfico e realizar uma regressão polinomial. R Core Team (2023).

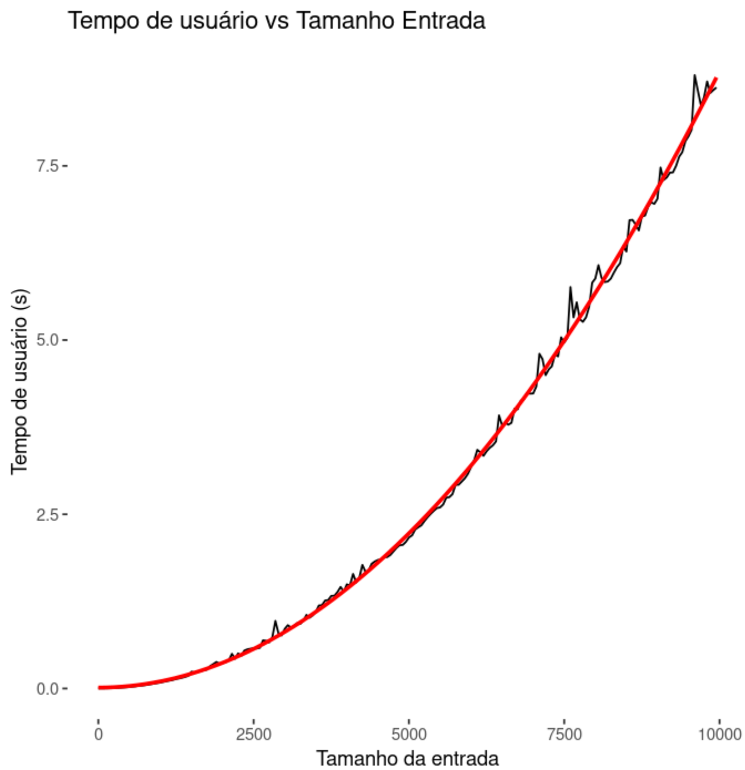


Figura 5: Curva obtida em testes

O comportamento da curva condiz com um programa de complexidade $O(n^2)$. Ainda sim, optamos por analisar uma regressão para os dados. Obtivemos a função descrita a seguir, com a não significância de β_0 e β_1 ao nível de $0,05$:

$$\hat{f}(n) = 8,830528 \cdot 10^{-8} n^2$$
$$R^2 = 0.9982$$

O tempo de sistema foi $0,0s$ para basicamente todas as execuções testadas. Tomando a média de tempo das execuções, obtemos um tempo de 0.000119985s. Realizando-se um teste-t sob $H_0 : \mu = 0$, não rejeitamos a hipótese nula e não temos evidência estatística de que a média seja diferente de zero para $\alpha = 0,05$. (*p-valor =0.0139*).

Tempo de usuário vs tempo de sistema

De acordo com Tanenbaum (2016), "quando um computador é multiprogramado, ele frequentemente tem múltiplos processos ou threads competindo pelo uso da CPU ao mesmo tempo". Se dois processos estão prontos para serem executados, quem decide quem será executado primeiro é o escalonador do sistema.

Sabemos que toda chamada de função ocasiona uma condição de corrida, gerando o escalonamento do processo e sua interrupção da execução pela CPU.

Dessa maneira, sabemos, por exemplo, que toda vez que a função *analisa_todos_os_pontos*() chama a função *contar_pontos_subjacentes*(), o nosso processo sai da CPU e aguarda até que o Sistema Operacional o coloque de novo em execução. O tempo de sistema mensura somente o tempo gasto na execução do programa, sem considerar o tempo de espera causado pelo escalonador do SO.

Uso de memória

Nosso programa faz $n + 2$ alocações da estrutura de dados Ponto. Cada Ponto contém 3 inteiros de *4 bytes*, o que totaliza *12 bytes* por alocação. Além das n alocações para armazenar os n pares ordenados, alocamos os pontos A e B.

Além disso, cada chamada da função *esta_dentro*() realiza 3 *allocs* provisórios da estrutura $Vetor$. Ao final da função, esses objetos são destruídos/desalocados da memória. De alguma maneira, é como se essa função tratasse essas variáveis como se fossem estáticas da pilha.

Usamos o utilitário *valgrind* para analisar possíveis *leaks* de memórias. A figura abaixo apresenta o resultado da nossa análise quando executamos um caso de teste de tamanho 1000.

```
==289108==
==289108== HEAP SUMMARY:
==289108==     in use at exit: 0 bytes in 0 blocks
==289108==   total heap usage: 1,500,499 allocs, 1,500,499 frees, 12,025,532 bytes allocated
==289108==
==289108== All heap blocks were freed -- no leaks are possible
==289108==
==289108== For lists of detected and suppressed errors, rerun with: -s
==289108== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
zonzin@rodrigo:~/Documentos/Faculdade/AEDSIII/tp1/tp1$
```

Figura 6: Uso da heap após execução

Como se pode constatar, todos os *allocs* foram desalocados e *"no leaks are possible"*. A alta quantidade de *allocs*, provavelmente se refere à função *esta_dentro*(), que aloca 3 vetores toda vez que é chamada (mas os desaloca logo no final de sua execução).

Se considerarmos que ela é chamada dentro de um for aninhado e a cada chamada ela faz 3 *allocs*, temos $3n^2$ *allocs* somente desta função. Consideramos ainda 1 *alloc* no *Selection Sort*, 1 *alloc* para a criação do *array de pontos*, n *allocs* para a cada ponto e 2 *allocs* para os pontos A e B. Definimos a função:

$$^{2}\ allocs(n) = 3n^2 + n + 3$$

Como testamos 1000 pontos, temos que $allocs(1000) = 3(1000)^2 + 1000 + 3 \Rightarrow allocs(1000) = 3\ 001\ 003$.

Não é o valor obtido, mas se o dividirmos por 2 encontramos $1\ 500\ 501,5$, que é o verdadeiro valor.

Isso nos sugere que a análise está incorreta por um termo $1/2$, mas que provavelmente segue a lógica correta.

Instruções para compilação e execução

Basta usar o comando *make* para efetuar a compilação e *makeremove* para limpar os arquivos intermediários. Para executar, digite:

./main -i suaEntrada.txt -o seuResultado.txt

Apêndice de dados

Todos as entradas testadas e arquivos *.csv* utilizados para a analise podem ser conferidos no seguinte repositório: `https://github.com/RodrigoZonzin/tp1-aeds3`.

[2] $n = tamanhodaentrada$

Referências

Anton, H. and Rorres, C. (2012). *Álgebra Linear com Aplicações*. Bookman, 10th edition.

Boulos, P. (2010). *Geometria Analítica: Um Tratamento Vetorial*. Pearson, 3 edition.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Algorithms*. The MIT Press, Cambridge, MA, 3rd edition.

Euclides (2013). *Os Elementos de Euclides*. Editora da Universidade de São Paulo, São Paulo, Brasil, 1st edition.

R Core Team (2023). *R: A Language and Environment for Statistical Computing*. Vienna, Austria.

Tanenbaum, A. S. (2016). *Sistemas Operacionais modernos*. Pearson, 4 th edition.