Lista de Exercícios 2 - CAP-241 2017

Prof. Dr. Gilberto Ribeiro de Queiroz.

Aluno: Paulo Henrique Barchi[1a]

13 de abril de 2017

[1]paulobarchi@gmail.com

[a] Laboratório Associado de Computação e Matemática Aplicada (LAC)
Coordenação de Laboratórios Associados (CTE)
Instituto Nacional de Pesquisas Espaciais (INPE)
São José dos Campos, SP - Brasil.

Exercício 01. A função mostrada abaixo computa o menor retângulo envolvente de um polígono simples. Faça a análise de complexidade do algoritmo básico envolvido nessa computação.

```
1 Rectangle ComputeMBR(const Polygon& poly)
2 {
3   const std::size_t nPts = poly.size();
4
5   if(nPts <= 3)
6     throw std::invalid_argument("Poligono inválido");
7
8   Rectangle r(poly[0].x,
9               poly[0].y,
10              poly[0].x,
11              poly[0].y);
12
13  for(std::size_t i = 1; i < nPts; ++i)
14  {
15    if(r.minx > poly[i].x) r.minx = poly[i].x;
16    if(r.miny > poly[i].y) r.miny = poly[i].y;
17    if(r.maxx < poly[i].x) r.maxx = poly[i].x;
18    if(r.maxy < poly[i].y) r.maxy = poly[i].y;
19  }
20
21  return r;
22 }
```

Solução. Em resumo, para todos os pontos do polígono passado como parâmetro, a função verifica as coordenadas x e y de cada ponto para estabelecer as coordenadas mínimas e máximas de x e y, que serão as coordenadas do menor retângulo envolvente do polígono em questão.

A análise dessa função se dá da seguinte forma:

1. Primeiramente, avaliamos as linhas 15, 16, 17 e 18 que estão dentro do anel do laço da linha 13. O tempo de execução de um comando de decisão é composto pelo tempo de execução dos comandos executados dentro do comando condicional, mais o tempo para avaliar a condição, que é *O(1)*. Consideramos que todos os condicionais serão avaliados, porém, a coordenada de um ponto não pode ser simultaneamente maior e menor que outra coordenada. Assim, neste caso, o tempo de execução das linhas 15, 16, 17 e 18 será dado por:

$$(O(1) + O(1)/2) + (O(1) + O(1)/2) + (O(1) + O(1)/2) + (O(1) + O(1)/2) =$$
$$O(max(O(1), O(1)/2, O(1), O(1)/2, O(1), O(1)/2, O(1), O(1)/2)) = O(1)$$

2. Agora, podemos seguir para a análise do laço da linha 13, que compreende as linhas 15, 16, 17 e 18. Quando temos um laço, o tempo é calculado como a soma do tempo de execução do corpo do laço, multiplicado pelo número de iterações do laço, mais o tempo de avaliar sua condição de terminação. Neste caso, o número de iterações do anel é $n-1$. Na etapa anteior, vimos que o corpo do anel é O(1), portanto, o tempo de execução do corpo do laço em todas as iterações é dado por:

$$(n-1) \times O(1) = O((n-1) \times 1) = O(n-1) = O(n)$$

Portanto, a função em questão é *O(n)*.

Exercício 02. É possível melhorar o algoritmo do *exercício 1*? Em que nível? Discuta.

Solução. Uma possível melhoria no algoritmo do *exercício 1* seria aproveitar a verificação de mínimo de cada coordenada para realizar a verificação de máximo de cada coordenada apenas caso a coordenada não seja mínima nesta iteração. Estas alterações foram feitas no código abaixo.

```cpp
Rectangle ComputeMBR(const Polygon& poly)
{
  const std::size_t nPts = poly.size();

  if(nPts <= 3)
    throw std::invalid_argument("Poligono inválido");

  Rectangle r(poly[0].x,
         poly[0].y,
         poly[0].x,
         poly[0].y);

  for(std::size_t i = 1; i < nPts; ++i)
  {
    if(r.minx > poly[i].x)
      r.minx = poly[i].x;
    else if(r.maxx < poly[i].x)
      r.maxx = poly[i].x;
    if(r.miny > poly[i].y)
      r.miny = poly[i].y;
    else if(r.maxy < poly[i].y)
      r.maxy = poly[i].y;
  }

  return r;
}
```

Com estas alterações, o algoritmo continuaria com o número de verificações $(4 \times nPts)$ para o pior caso, e, na notação de **big-*O***, continuaria a ser *O(n)*. No entanto, é uma alternativa mais elegante, pois faz uso das verificações feitas previamente e fica com uma lógica mais consistente, pois caso um valor seja o mínimo, ele só seria também o máximo caso todos os valores sejam iguais, o que torna desnecessária a nova atribuição.

Exercício 03. O **envoltório convexo** ou **fecho convexo** de um conjunto de pontos no espaço bidimensional corresponde ao **polígono convexo** formado por pontos do conjunto P que contém todos os pontos de P.

O Algoritmo 1.3 (apresentado no documento da Lista) computa os pontos do envoltório convexo para um conjunto de pontos de entrada P qualquer. Pede-se:

- Implemente esse algoritmo em C++ padrão. Você poderá utilizar os componentes da biblioteca padrão de C++ que julgar necessário nessa implementação. Mas deverá justificar a escolha dos componentes e algoritmos.

- Faça um programa que tome o tempo, em segundos/minutos/horas, para a execução desse algoritmo para um conjunto de pontos de tamanho distintos. Sugestão: (a) crie uma função que gere de forma aleatória um conjunto de pontos de um determinado tamanho; (b) use a biblioteca <chrono> de C++; (c) mostre na forma de uma tabela e um gráfico esse desempenho.

Para verificar se um ponto r se encontra a esquerda ou direita de um dado segmento $\overline{pq}$ você deverá calcular o produto vetorial dos vetores formados por esses pontos. O sinal do resultado do produto vetorial lhe dará essa informação:

$$produto = (q.x - p.x) \times (r.y - p.y) - (q.y - p.y) \times (r.x - p.x)$$

Solução. O código referente à implementação em `C++` do Algoritmo 1.3, fornecido no documento da Lista 2, está no arquivo `exercicio03.cpp`, o qual está compactado no mesmo arquivo que este documento (`lista2.zip`).

Para esta implementação, reutilizei a classe `point` implementada para a *Lista 1* com algumas alterações. Os métodos desnecessários foram removidos e foram implementados os métodos de comparação `==` e `<` para que o `sort` da biblioteca padrão de `C++` funcionasse devidamente. Também foi adicionado um método para cálculo do produto vetorial deste ponto (`this`) com os pontos do segmento de reta passados como parâmetro.

O método `removeDuplicates` faz uso do `sort` já mencionado e do `erase` da classe `vector` para remoção de registros (pontos, neste caso) duplicados. Como orientado, a biblioteca `<chrono>` foi utilizada para obter os tempos de processamento. E também, a geração de pontos aleatórios foi feita com uso de `srand`.

Conforme solicitado, a tabela e o gráfico abaixo apresentam o desempenho do código implementado para diferentes tamanhos dos vetores de pontos, com o tempo de execução. O tamanho do conjunto de pontos é iniciado com o valor 10, e, por 10 iterações, este tamanho é dobrado. Para melhor visualização, o gráfico foi plotado com escala logaritmica no eixo horizontal.

n	10	20	40	80	160	320	640	1280	2560	5120
time (s)	4.602e-05	0.000143919	0.00059525	0.00303475	0.0121082	0.0561292	0.202083	1.2074	9.34098	73.7442

Tabela 1: Tempo de processamento para diferentes números de pontos.

Exercício 04. Faça a análise de complexidade do algoritmo do *exercício 3*.

Solução.

1. Primeiramente, avaliamos as linhas 7, 8 e 9 que estão dentro do anel do laço mais interno da linha 6, que por sua vez está dentro do laço externo da linha 3. O tempo de execução de um comando de decisão é composto pelo tempo de execução dos comandos executados dentro do comando condicional, mais o tempo para avaliar a condição, que é *O(1)*. Assim, o tempo de execução das linhas 7, 8 e 9 será dado por:

$$(O(1) + O(1) + (O(1)) = O(max(O(1), O(1), O(1))) = O(1)$$

2. Agora, podemos seguir para a análise do laço interno da linha 6, que compreende as linhas 7, 8 e 9. Quando temos um laço, o tempo é calculado como a soma do tempo de execução do corpo do laço, multiplicado pelo número de iterações do laço, mais o tempo de avaliar sua condição de terminação. Neste caso, o número de iterações do anel é n. Na etapa anteior, vimos que o corpo do anel é O(1), portanto, o tempo de execução do corpo do laço em todas as iterações é dado por:

$$(n) \times O(1) = O(n \times 1) = O(n)$$

3. Com isso, podemos seguir para a análise do laço mais externo da linha 3, que compreende o laço interno da linha 6 e as linhas 7, 8 e 9. Da mesma forma, calculamos o laço como a soma do tempo de execução do corpo do laço, multiplicado pelo número de iterações do laço, mais

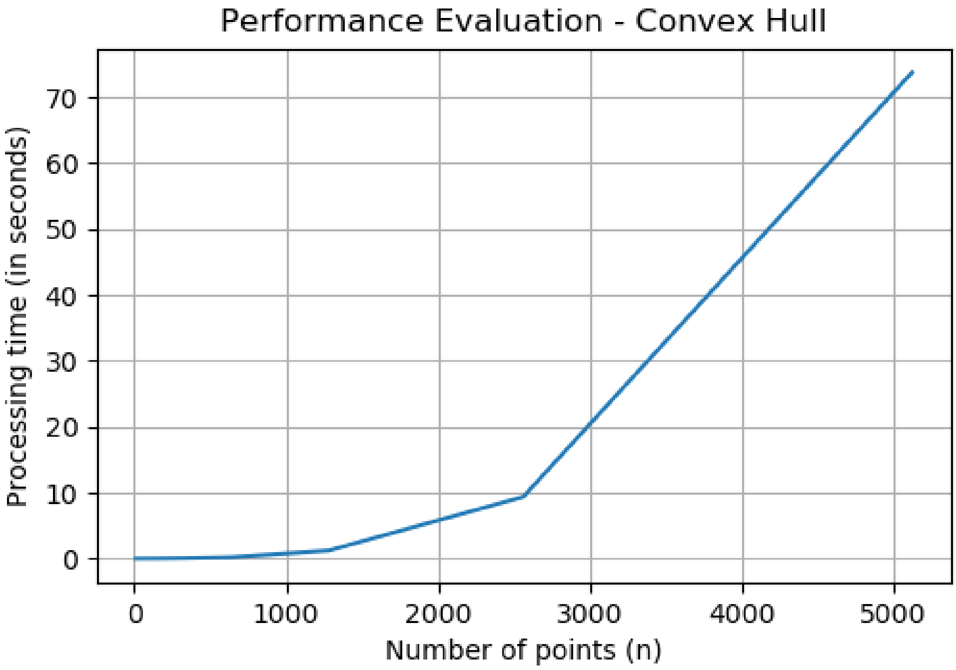

Figura 1: Análise de desempenho - Envoltório Convexo.

o tempo de avaliar sua condição de terminação. Neste caso, a linha 3 define um laço para todos os pares possíveis de pontos, com a verificação de que este par não consista do ponto associado com ele mesmo. Então, temos *O(1)* para a verificar se os pontos são diferentes. Como o laço é para todos os pares possíveis de pontos, para implementação são necessários dois laços aninhandos, cada um com *O(n)*. Assim, decompondo este laço externo em dois laços aninhandos, considerando a verificação dos pontos e o laço interno, temos:

$$\sum_{i=1}^{n}\sum_{i=1}^{n} n = n \times n \times n = O(n^3)$$

O Processo de criação de V (conjunto ordenado dos vértices do envoltório convexo) a partir de E necessitaria de uma inspeção na implementação do algoritmo utilizado para realizar a ordenação. No entanto, esse processo é garantidamente menos custoso que $O(n^3)$. Portanto, como consideramos sempre os máximos para obtenção do **big-*O***, a função em questão é $O(n^3)$.