Algoritmos e Estrutura de Dados I

Lorran Marques
Escola Politécnica - PUCRS

August 23, 2022

Abstract

Quando desenvolvemos um algoritmo, devemos pensar não apenas em escrever um algoritmo correto, mas também em escrever um algoritmo eficiente. Isto é, dois algoritmos podem resolver o mesmo problema com custos computacionais e temporais distintos. Nesse trabalho foram propostos seis algoritmos e, utilizando a contagem de operações, determinamos o custo de cada um.

1 Introdução

Para conseguirmos entender se o nosso código é eficiente, podemos nos munir de técnicas para determinar o quão dependente a execução do código é do tamanho do problema. Em outras, se assumirmos um vetor N e verificarmos que o performa um número X de operações. Quantas operações serão necessárias para executar o mesmo algoritmos com um vetor de tamanho $100N$? E de tamanho $1000N$?.
Nesse trabalho utilizamos o método de contagem de operações para determinarmos o custo do algoritmo: para cada função, vamos adicionar uma variável *cont_op* que é incrementada no início da função e com isso saberemos quantas chamadas de função serão realizadas.

2 Método

2.1 Teoria

Durante a história da informática descobrimos que as funções típicas para o gasto de operações de um algoritmo separam-se em dois grandes grupos:

Funções polinomiais Nestas funções a variável (ou variáveis) representando o tamanho do problema é usada em um polinômio, como por exemplo:

$$f(n) = n^3 + 4n + 16 \tag{1}$$

Nessas funções, o expoente mantém-se inalterado independentes da base n. No entanto, não há limites para o tamanho do expoente, pode ser um expoente baixo, como n^1, ou um expoente como n^{300}. Note também que, devido ao comportamento das funções polinomiais, para compararmos o custo entre dois algoritmos, podemos levar em consideração apenas o n de maior expoente, isso porque, no exemplo da função 1:

$$\lim_{n \to +\infty} n^3 + 4n + 16 \approx \lim_{n \to +\infty} n^3 \tag{2}$$

Em outras palavras, podemos dizer que o expoente maior cresce mais depressa que os outros expoentes.

Funções exponenciais Nestas funções as variáveis representando o tamanho do problema são usadas **como expoentes**, como por exemplo

$$f(n) = 3 * 2^n + 11. \tag{3}$$

A consequência é que estas funções costumam crescer muito mais depressa do que as polinomiais, e se o algoritmo consumir operações (ou tempo) de uma forma exponencial temos um problema sério nas mãos.

2.2 Procedimento

O primeiro passo para determinarmos o custo do nosso algoritmo seria determinar qual o tipo da função que ele segue. Para isso podemos fazer o seguinte procedimento:

1. Calculamos a quantidade de operações *cont_op* para cada n. Nesse trabalho utilizamos $1 \leq n \leq 100$, somados um a um. Exceto para os algoritmos 4 e 6, devido a seu alto custo, nesses dois casos utilizamos $1 \leq n \leq 35$.

2. Plotamos um gráfico de $n \times \log_{10} cont\_op$

3. Caso o gráfico gerado seja uma reta, a função é exponencial.

4. Caso contrário, plotamos o gráfico de $\log_{10} n \times \log_{10} cont\_op$. Caso o resultado seja uma reta, a função é polinomial

Após determinado o tipo de função devemos calcular o quão depende ela é de n.

Para uma função **exponencial**, como ela é do tipo $f(n) = a * b^n$, queremos determinar o valor de b. Portanto, devemos de alguma forma isolar a variável b. Manipulando a equação, aplicando logaritmos em ambos lados, temos que:

$$\log_{10} f(n) = \log_{10} a * b^n = \log_{10} a + n * \log_{10} b \tag{4}$$

Como a função acima se comporta como uma reta, podemos calcular a inclinação da reta r através de:

$$r \approx \frac{f(x_2) - f(x_1)}{x_2 - x_1} \approx \frac{\log_{10} f(x_2) - \log_{10} f(x_1)}{x_2 - x_1} \tag{5}$$

Como $f(n) = cont\_op$, podemos determinar r com base em dois valores distintos da lista de *cont_op* que calculamos. Assim sendo:

$$r \approx \frac{\log_{10} cont\_op_2 - \log_{10} cont\_op_1}{x_2 - x_1} \tag{6}$$

Por fim, como $r = log_{10} b$ temos que:

$$b \approx 10^r \tag{7}$$

Para uma função **polinomial**, a reta apenas aparece quando é aplicado o logaritmo em **ambos** os eixos. Sendo assim, repetindo a manipulação utilizada no caso das exponenciais, teremos que:

$$b \approx r \approx \frac{\log_{10} f(x_2) - \log_{10} f(x_1)}{\log_{10} x_2 - \log_{10} x_1} \tag{8}$$

2.3 Software

Os algoritmos propostos foram reescritos em **Python** e a biblioteca **PyPlot** foi utilizada para gerar os gráficos. A seguinte função foi criada para executar os algoritmos de 1 a 5:

```
def calcFunction(f,min,max,step):
    global cont_op
    n = []
    values = []
    for i in range(min,max,step):
        cont_op = 0
        print(i,f(i),cont_op)
        n.append(i)
        values.append(cont_op)
    return n,values
```

Os valores de n e *cont_op* retornados foram armazenados em um arquivo .txt para posterior consulta.

O algoritmo 6, por necessitar de dois parâmetros, foi executado separadamente.

As funções para calcular o b, nos casos exponencial e polinomial, respectivamente, são as seguintes:

```
def calcBExpo(n,op):
    x1 = n[0]
    x2 = n[-1]
    y1 = op[0]
    y2 = op[-1]
    r = ( ( y2 ) - (y1) ) / ( x2 - x1 )
    b = 10**r
    return b

def calcBPolinomial(n,op):
    x1 = n[0]
    x2 = n[-1]
    y1 = op[0]
    y2 = op[-1]
    r = ( ( y2 ) - (y1) ) / ( ( x2) - (x1) )
    return r
```

Importante notar que nas chamadas dessas funções os valores de n e op já são passados com os logaritmos aplicados quando necessário, conforme trecho abaixo, onde é feita a leitura do arquivo com os dados de contagem de operações e a plotagem dos gráficos e cálculo do b nos algoritmos de 1 a 5:

```
def main(algoritmo):
    caminho = "Algoritmo "+str(algoritmo)

    f = get_file(caminho+".txt")
    n = [int(i[0]) for i in f]
    op = [int(i[1]) for i in f]
    ax = plt.scatter(n,op)
    plt.show()
    ax.figure.savefig(caminho+" - original.png")
    plt.clf()

    op = [math.log10(i) for i in op[3:]]
    n = n[3:]
    axlog = plt.scatter(n,op)
    axlog.figure.savefig(caminho+" - logy.png")
    plt.show()
    plt.clf()
    bexpo = calcBExpo(n,op)

    n = [math.log10(i) for i in n]
    axlog = plt.scatter(n,op)
    axlog.figure.savefig(caminho+" - logxlogy.png")
    plt.show()
    plt.clf()

    bpoli = calcBPolinomial(n,op)

    return bexpo,bpoli
```

O algoritmo 2 possui $cont\_op = 0$ para $n = 1, 2, 3$, sendo assim, utilizamos, como valor para *x1*, *n[3]*. O algoritmo 6 foi executado à parte.

3 Resultados

Os gráficos resultantes para cada algoritmo, assim como o arquivo com os dados de n e cont_op, encontram-se anexos à este relatório. Os nomes dos gráficos seguem a seguinte lógica:

Algoritmo *número do algoritmo* - original.png → É o gráfico com os eixos sem logaritmos.
Algoritmo *número do algoritmo* - logy.png → É o gráfico com o logaritmo aplicado **apenas** ao eixo y.
Algoritmo *número do algoritmo* - logxlogy.png → É o gráfico com o logaritmo aplicado **em ambos eixos**.

Analisando-os, podemos notar que todos os algoritmos, à exceção do **Algoritmo 6** são polinomiais. Sendo assim, aplicando o descrito na Seção 2.2, obtemos os seguintes valores de b

Algoritmo	b
Algoritmo 1	3.92
Algoritmo 2	2.29
Algoritmo 3	1.97
Algoritmo 4	6.10
Algoritmo 5	3.99
Algoritmo 6	1.63

4 Conclusão

Com a Tabela 3 obtida na seção anterior concluimos as seguintes funções para os algoritmos:

Algoritmo 1

$$f(n) \propto n^{3.92} \tag{9}$$

Algoritmo 2

$$f(n) \propto n^{2.29} \tag{10}$$

Algoritmo 3

$$f(n) \propto n^{1.97} \tag{11}$$

Algoritmo 4

$$f(n) \propto n^{6.10} \tag{12}$$

Algoritmo 5

$$f(n) \propto n^{3.99} \tag{13}$$

Algoritmo 6

$$f(n) \propto 1.63^{n} \tag{14}$$