Uma metodologia para cálculo da complexidade de algoritmos

Laira V. Toscani

Departamento de Informática - UFRGS
Atualmente na Universidade Nova de Lisboa
Fac. de Ciência e Tecnologia - Depto. de Informática
Grupo de Lógica Computacional
2825 - Monte da Caparica - Portugal
e-mail: lvt@host.fctunl.rccn.pt

Paulo A. S. Veloso

Departamento de Informáuca - PUC/RJ
R. Marquês de São Vicente 225
22453 - Rio de Janeiro - Brasil
e-mail: uservelo@lncc

Resumo

As complexidades induzidas por estruturas algorítmicas são identificadas e expressas por equações. Com base nessas expressões de complexidade é proposta uma metodologia para o cálculo da complexidade de algoritmos.

1. Introdução

A análise da complexidade de um algoritmo é realizada de maneira muito particular, o que é compreensível, já que a complexidade é uma medida que tem parâmetros bem particulares do algoritmo, como a função tamanho da entrada e as operações fundamentais. Apesar disto alguns conceitos são gerais, no sentido que só dependem da estrutura do algoritmo. São esses conceitos gerais que serão explorados neste artigo.

O objetivo deste trabalho é, pois, apresentar as complexidades intrínsecas que possuem as estruturas algorítmicas e propor uma metodologia de análise da complexidade de algoritmos baseada nessas complexidades intrínsecas.

2. Definição de Complexidade

A complexidade de tempo de um algoritmo, ou simplesmente a complexidade como será chamada neste trabalho, é a quantidade de trabalho dispendida pelo algoritmo.

Para medir esta quantidade de trabalho é escolhida uma operação chamada operação fundamental e então é contado o número de execuções dessa operação na execução do algoritmo.

Eventualmente podem ser consideradas mais de uma operação fundamental, com pesos diferentes.

Esta contabilização de execuções da operação fundamental não é constante para um algoritmo, pois depende da entrada. É função, em geral, da quantidade de informação da entrada. A medida da quantidade de informação da qual depende a complexidade, por sua vez, varia de problema para problema e é chamada tamanho da entrada ou tamanho do problema.

A medida da complexidade de um algoritmo é portanto uma medida particular, porque depende da escolha da operação fundamental e do critério usado no cálculo do tamanho da entrada. Só faz sentido, portanto, comparar complexidades de algoritmos de uma mesma classe (mesma operação fundamental e mesma função tamanho da entrada). Realmente, o interessante é comparar algoritmos que resolvem um mesmo problema e em geral esses algoritmos estão na mesma classe. Além disso, a complexidade de um algoritmo pode depender não só do tamanho da entrada, mas também da entrada em particular.

Chamando E_n o conjunto das entradas de tamanho n, tomando $e \in E_n$ e chamando c(e) o número de execuções da operação fundamental, tem-se (para a entrada e) :

$$CPC(n) = \max_{e \in E_n} c(e)$$

que é chamada complexidade no pior caso. Outras medidas de complexidade também podem ser usadas, mas neste trabalho vamos tratar somente da complexidade no pior caso.

Um algoritmo, segundo [KNU 83] , é um método abstrato que computa a solução de um problema. Um problema ([VEL & VEL 81]) é uma terna $p = < D, R, q >$, onde D é o domínio de dados, R o domínio de resultados e q uma relação de D em R que define o problema. A solução de um problema $p = < D, R, q >$ é uma função $\alpha : D \rightarrow R$ tal que $(\forall d \in D)((d, \alpha(d)) \in q)$. Logo, dado um problema $p = < D, R, q >$ e uma solução α para p, um algoritmo a_α é um método abstrato que computa α.

Dado um problema $p = < D, R, q >$, com solução α, seja $\mathfrak{A}$ o conjunto dos algoritmos que computam α. Para calcular a complexidade de um algoritmo $a \in \mathfrak{A}$ é necessário escolher a operação fundamental (ou operações fundamentais) e definir a função que dá o tamanho da entrada. Se houver mais de uma operação fundamental é necessário definir o peso de cada uma. Seja E o conjunto de todas as seqüências de execuções de operações fundamentais. Tem-se então o seguinte diagrama:

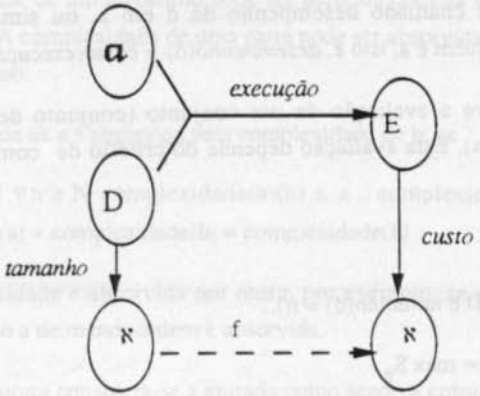

Figura 1

Onde:

- *execução* : $\mathfrak{A}$ x D → E : *execução*(a, d) := seqüência de execuções de operações fundamentais efetuadas na execução do algoritmo a, com entrada d.

- *custo* : E → ℵ ; *custo*(s) := comprimento da seqüência s, definido conforme o peso estabelecido para as operações fundamentais.

- *tamanho* : D → ℵ ; *tamanho*(d) := tamanho da entrada d .

Escolhidas as funções fundamentais e *tamanho*, o conjunto $\mathfrak{A}$ é particionado (ficam na mesma classe os algoritmos com mesmas funções fundamentais e mesma função *tamanho*). A complexidade é então definida dentro de cada classe. Considere A como uma dessas classes e defina a complexidade de um elemento de A, como uma função do tipo ℵ → ℵ .

A *complexidade* é portanto um funcional do tipo A → (ℵ → ℵ), isto é *complexidade* : A → f e f:={f/f: ℵ → ℵ }. Para a ∈ A, *complexidade*(a) = f, f: ℵ → ℵ .

Note que a função *tamanho* não é inversível. Dado n ∈ ℵ , quem é f(n)? Para avaliar f(n) é necessário considerar todas as entradas d ∈ D com *tamanho*(d) = n, calcular *custo*(*execução*(a, d)) e então conforme o critério de complexidade desejado estabelecer f(n). Assim pode-se dizer que:

f(n):= *avaliação*({*custo*(*execução*(a, d))/ d ∈ D e *tamanho*(d) = n}).

E *custo*(*execução*(a, d)) pode ser chamado desempenho de d em a, ou simplesmente desempenho de d, quando está claro quem é a, isto é, *desempenho*(d):= *custo*(*execução*(a, d)).

O cálculo de f(n), portanto, envolve a avaliação de um conjunto (conjunto de todos os desempenhos de dados de tamanho n). Esta avaliação depende do critério de complexidade desejado.

Dados a $\in$ $\mathfrak{a}$ e n $\in$ $\aleph$, seja

$$S_n := \{ \textit{desempenho}(d) / d \in D \text{ e } \textit{tamanho}(d) = n \},$$

$$CPC(n) := \textit{avaliaçãoCPC}\ (S_n) := \max S_n$$

Usualmente o algoritmo é construido e depois analisado quanto à complexidade. O processo entretanto, seria mais eficaz se a complexidade fosse um fator integrante do projeto do algoritmo. Uma metodologia de cálculo de complexidade poderia incentivar esta prática. Mas, apesar da complexidade no pior caso ser uma medida muito particular à classe A do algoritmo que está sendo analisado (pois, depende da função *tamanho* e das operações fundamentais), alguns aspectos do cálculo da complexidade não dependem do que faz o algoritmo mas somente da sua estrutura e assim podem ser generalizados.

Assim, uma metodologia de cálculo de complexidade para estruturas algorítmicas pode ser desenvolvida.

3. Complexidade das Estruturas Algorítmicas

Serão estudadas, uma a uma, as seguintes principais estruturas algorítmicas:

a ← **b**;

a ; **b**;

if **a** then **b** else **c**;

for k = **i** to **j** do **a**;

while **a** do **b**;

Notação: c(**a**) = complexidade de **a** no pior caso.

A complexidade de um algoritmo pode ser definida como a soma das complexidades das suas várias partes. A complexidade de uma parte pode ser absorvida pela complexidade de outra parte no seguinte caso:

A complexidade de **a** é absorvida pela complexidade de **b**, se

$\exists c\ \exists N\ \forall n \geq N$ complexidade(**a**)(n) $\leq c$. complexidade(**b**)(n). E pode-se dizer que complexidade(**a**) + complexidade(**b**) = complexidade(**b**).

Uma complexidade é absorvida por outra, por exemplo, se ambas são polinômios de mesmo tipo, neste caso a de menor ordem é absorvida.

Para cada estrutura considera-se a entrada como sendo a entrada corrente do algoritmo, quando a estrutura é executada. A saída é a entrada corrente, na saída da estrutura.

A notação $f = f_1 \rightarrow g_1, g_2$ será usada para identificar

$$f = \begin{cases} g_1 & \text{se } f_1 \\ g_2 & \text{caso contrário} \end{cases}$$

3.1. Atribuição

A atribuição é uma estrutura sintaticamente simples, cuja semântica depende do tipo de dado atribuído.

a ← **b**

A complexidade da atribuição também depende do tipo da atribuição:

$$c(\mathbf{a} \leftarrow \mathbf{b}) = c(\leftarrow) + c(\mathbf{b}) \qquad (3.1\text{-}1)$$

onde $c(\leftarrow)$ é a complexidade da atribuição para o tipo do dado **a**. Assim, $c(\leftarrow)$ pode ser a complexidade da inserção de um nodo num grafo se **a** é tipo grafo, ou a atualização de uma matriz se **a** é uma matriz, etc... .

3.2. Seqüência

Esta estrutura simples tem também uma equação de complexidade muito simples.

a ; **b**

A complexidade da seqüência é a soma das complexidades dos componentes, isto é:

$$c(\mathbf{a}\,;\,\mathbf{b})(n) := c(\mathbf{a})(n) + c(\mathbf{b})(n') \qquad (3.2\text{-}1)$$

onde n' é o tamanho da entrada após a execução de **a** dado que antes da execução era n.

A ordem de execução de uma seqüência é importante no cálculo da complexidade, a qual pode variar sensivelmente com uma alteração dessa ordem. A complexidade varia quando a execução de **a** altera o tamanho do problema sobre o qual **b** vai atuar. Um exemplo: **a** e **b** atuam sobre uma lista, a execução de **a** reduz o tamanho da lista e **b** faz uma procura seqüêncial na lista.

Muitas vezes, a complexidade de **a** é absorvida pela de **b**, ou vice-versa.

3.3. Condicional

A estrutura condicional pode tomar várias formas, sendo aqui adotada a mais usual:

if **a** then **b** else **c**

A complexidade associada ao if é a complexidade da avaliação de **a** mais a complexidade de **b** ou a complexidade de **c**. Esta escolha depende também do critério de complexidade.

$$c(\text{if } \mathbf{a} \text{ then } \mathbf{b} \text{ else } \mathbf{c}) := c(\mathbf{a}) + \max(c(\mathbf{b}), c(\mathbf{c}))$$

Entende-se como c(**a**) a complexidade da avaliação de **a**.

A CPC define max como:

$$\max(f, g) = \quad (\exists N)(\forall n \geq N)\ f(n) > g(n) \rightarrow f,$$
$$(\exists N)(\forall n \geq N)\ g(n) > f(n) \rightarrow g\ ,\ h \qquad (3.3\text{-}1)$$

com $h(n) = (f(n) > g(n)) \rightarrow f(n), g(n)$.

Usualmente, depois de um certo n, f(n) é sempre maior que g(n), ou vice-versa, e max(f, g) = f ou g, respectivamente. Entretanto pode acontecer, como no caso abaixo, de max(f, g) = h. Imaginemos, por exemplo, um algoritmo que manipula grafos e efetua duas operações, uma cuja complexidade depende exclusivamente do número de arestas e outra cuja complexidade varia exclusivamente com o número de nodos. A função tamanho da entrada, do algoritmo, tem duas componentes, número de arestas e número de nodos, combinados de alguma forma (somados, por exemplo). O algoritmo constitui-se de um condicional cujo ramo then efetua uma das operações, por exemplo aquela dependente do número de arestas e o ramo else efetua a outra operação. Aumentando o número de arestas somente aumenta a complexidade do ramo then e aumentando o número de nodos somente aumenta a complexidade do ramo else. Assim,

aumentando convenientemente a entrada, a complexidade de cada ramo pode superar a do outro e conseqüentemente max(f, g) = h. Nesse caso a CPC tomará o máximo pontual.

Se a estrutura condicional não possui a parte else a complexidade fica simplificada:

c(if **a** then **b**):= c(**a**) + c(**b**).

3.4. Iteração

A iteração também é uma estrutura simples para o cálculo da complexidade.

for k = i to j do **a**

A complexidade da iteração tem dois casos a se considerar:

a) Se n não varia durante a execução de **a**, tem-se:

$$c(\text{for } k = i \text{ to } j \text{ do } a) = (j-i+1) \cdot c(a) \qquad (3.4\text{-}1)$$

b) Se entretanto o tamanho da entrada varia com a execução de **a** tem-se:

$$c(\text{for } k = i \text{ to } j \text{ do } a)(n) = \sum_{k=0}^{j-i} c(a)(t^k(n)) \qquad (3.4\text{-}2)$$

onde t(n) = "o tamanho da entrada após a execução de **a** dado que antes da execução era n", $t^{k+1}(n) = t^k(t(n))$ e $t^0(n) = n$.

É bem comum o caso b), com t(n) = n e c(**a**) um polinômio de ordem k. Então

$$c(\text{for } k = i \text{ to } j \text{ do } a)(n) = O(n^j).$$

Outro caso freqüente é aquele em que, $c(a) = O(n^r)$, t(n) = n-c (isto é, **a** tem complexidade polinomial de ordem r e o tamanho do problema decresce de uma constante c), i = 1 e j = n. Então

$$c(\text{for } k = i \text{ to } j \text{ do } a)(n) = \sum_{k=0}^{n} (n-ck)^r = O(n^{r+1})$$

Se j - i independe de n, tem-se um somatório com um número fixo de parcelas. No caso mais usual a maior parcela absorve as demais e define a complexidade da iteração.

3.5. Iteração Condicional

Exemplificaremos com um tipo de iteração condicional. Outros tipos seriam tratados de forma similar.

while a do b

A complexidade do while é definida indutivamente como a complexidade da avaliação de **a** e, se **a** é **true**, mais a complexidade de **b** e mais a complexidade do while com a nova entrada resultante da execução do **b**.

$$c(\text{ while a do b })(n) := c(\mathbf{a})(n) + \{\neg\, \mathbf{a} \rightarrow 0, [\, c(\mathbf{b})(n) + c(\text{ while a do b })(t(n))]\},$$

onde t(n) = "o tamanho da entrada após a execução de **b**, dado que antes da execução era n".

$$c(\text{ while a do b })(n) := [c(\mathbf{a}) + c(\mathbf{b})](n) + [c(\mathbf{a}) + c(\mathbf{b})]\,(t(n)) + \ldots + c(\mathbf{a})\,(t^k(n))$$

$$:= c(\mathbf{a})\; t^k(n) + \sum_{i=0}^{k-1} [c(\mathbf{a}) + c(\mathbf{b})]\,(t^i(n)) \qquad (3.5\text{-}1)$$

com k = max S(n) e

S(n) = {k / existe uma seqüência de entradas sucessivas da estrutura de

tamanhos n, t(n), t(t(n)),. . . , $t^k(n)$, tal que avaliaram **a** como **true**,

exceto na última entrada da seqüência}.

A expressão (3.5-1) maximiza o número de vezes que o while é executado e maximiza as complexidades de **a** e **b**.

Em geral c(**a**) é absorvida por c(**b**) e então tem-se:

$$c(\text{ while a do b })(n) := c(\mathbf{a})\; t^k(n) + \sum_{i=0}^{k-1} c(\mathbf{b})\, t^i(n);$$

se além disso $t^k(n)$ é uma constante a equação fica assim:

$$c(\text{ while a do b })(n) := \sum_{i=0}^{k-1} c(\mathbf{b})\, t^i(n),$$

e os mesmos casos da iteração incondicional se apresentam.

4. Conclusões

Foram analisadas algumas estruturas algorítmicas, as mais usadas na programação imperativa, quanto a complexidade por elas geradas.

Às estruturas foram associadas equações de complexidade, em geral bastante simples. Entretanto, alguns cuidados são importantes e podem afetar sensivelmente a aplicação das equações. O acompanhamento da variação do tamanho do problema em cada aplicação de uma equação é um cuidado importante. Cada estrutura, como foi visto, tem suas particularidades às quais se deve estar atento. A complexidade da atribuição depende do tipo do dado envolvido. No condicional, às vezes é necessário considerar o máximo pontual das complexidades das partes then e else. Na iteração é importante verificar se o tamanho do problema varia durante a execução, pois a não variação simplifica o cálculo da complexidade. Na iteração condicional, a equação (3.5-1) faz uma estimativa muito pessimista da complexidade, talvez exagerada. Esta estimativa pode ser suficiente, por exemplo, se esta complexidade é absorvida em outro passo. Entretanto, se for necessário uma maior precisão, a complexidade pode ser ajustada através de uma análise individualizada do caso.

Assim, fica definida uma metodologia para cálculo de complexidade que deve ser um incentivo à análise da complexidade de algoritmos.

Estruturas algorítmicas mais gerais como Divisão e Conquista e Programação Dinâmica foram já analisadas em outros trabalhos [TOS & VEL 85] e [TOS & VEL 86].

Este é mais um trabalho no sentido de facilitar a análise da complexidade de algoritmos e antecipá-la tanto quanto possível para a fase de desenvolvimento do algoritmo.

Referências

[KNU 83] KNUTH, D. E. "Algorithm and Program: Information and Data". Communication of ACM, New York, 26(1):56, jan. 1983. pp 56.

[TOS & VEL 85] TOSCANI, L. V. & VELOSO, P. A. S. Uma Especificação Formal para a Programação Dinâmica. In: Congresso da Sociedade Brasileira de Computação, 5. , Porto Alegre, jul. 20-27, 1985. Anais. Porto Alegre, SBC/CLEI/UFRGS, 1985. p.477-86.

[TOS & VEL 86] TOSCANI, L. V. & VELOSO, P. A. S. Divisão e Conquista: análise da complexidade. In: Seminário Integrado de Software e Hardware, 13., Olinda, jul. 19-25, 1986. Anais. Recife, SBC/UFPE, 1986. p. 89-104.

[TOS & VEL 87] TOSCANI, L. V.& VELOSO, P. A. S. Análise da Complexidade de Programas Abstratos. In: Congresso Nacional de Matemática Aplicada e Computacional 10, Gramado/RS, set. 21-25, 1987. Anais. Porto Alagre, SBMAC,1987. p. 978-83.

[TOS 88] TOSCANI, L. V. "Métodos de Desenvolvimento de Algoritimos: Especificação Formal, Análise Comparativa e de Complexidade". Rio de Janeiro, Depto. de Informática da PUC/RJ. 1988. Tese de Doutorado.

[VEL & VEL 81] VELOSO, P. A. S. & VELOSO, S. R. M. "Problem Decomposition and Reduction: Applicability, Soundness, Completeness". In: Progress in Cybernetics and Systems Research. Washington, Hemisphere, 1981. v.8.