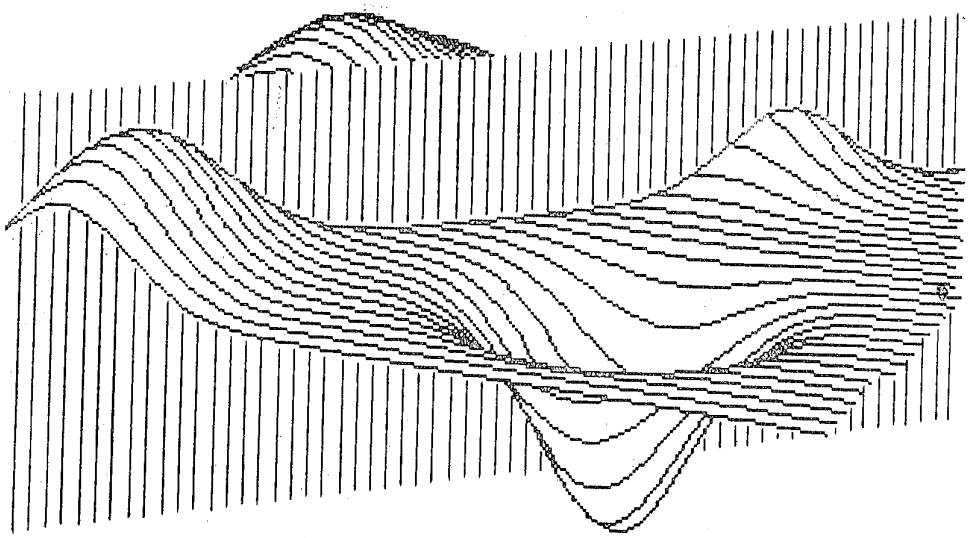
X Congresso Nacional de Matemática Aplicada e Computacional

21 A 25 DE SETEMBRO DE 1987
SERRANO CENTRO DE CONVENÇÕES – GRAMADO - RS



X CNMAC

Rt ...os

VOL. 2

ANÁLISE DA COMPLEXIDADE DE PROGRAMAS ABSTRATOS

LAIRA VIEIRA TOSCANI
Curso de Pós-Graduação em Ciência da Computação-UFRGS
Caixa Postal 1501 90.001 - Porto Alegre - RS

PAULO AUGUSTO S. VELOSO
Departamento de Informática/PUC-RJ
Rua Marques de São Vicente,225
22453 - Rio de Janeiro - RJ

Introdução

A complexidade de um algoritmo é medida em função do "volume" de dados, ao que se chama tamanho do problema. Por exemplo, se o dado é uma matriz, o tamanho do problema é a dimensão da matriz, se o dado é uma lista, é o número de elementos da lista.

A partir de um programa abstrato é possível estudar-se a complexidade de uma classe de algoritmos, a classe dos algoritmos provenientes do programa abstrato pela substituição dos predicados e funções abstratas por predicados e funções executáveis.

Métodos de desenvolvimento de algoritmos para resolução de problemas, podem ser especificados como programas abstratos. Veloso em [1] definiu a Divisão e Conquista como um programa abstrato, dando a especificação formal por meio de Tipos Abstratos de Dados. A partir desse, outros trabalhos foram desenvolvidos e outros métodos de desenvolvimento de algoritmos especificados formalmente como em [2] e [3].

Especificação Formal de Métodos de Desenvolvimento de Algoritmos

A formalização de métodos computacionais busca um aproveitamento mais eficaz dos métodos, através de uma melhor compreensão e maior clareza, além de facilitar o estudo de seus desempenhos.

As especificações formais apresentadas nos trabalhos mencionados na secção anterior consistem em um diagrama sintático que define o domínio das funções e predicados, um programa abstrato que dá a estru

tura algorítmica do método e uma axiomatização, que define o interrelacionamento das funções e predicados que compõem o método, além de uma verificação de correção do programa abstrato. Uma instância da abstração é um algoritmo definido pela instanciação do programa abstrato por funções e predicados que satisfazem os axiomas da especificação.

A complexidade do programa abstrato é calculada em função da complexidade das funções e predicados primitivos (funções e predicados que compõem a abstração). A complexidade de uma instância dessa abstração é a complexidade de programa abstrato, tendo como funções e predicados primitivos os especificados na definição da instância.

Neste trabalho será estudada sòmente a complexidade de tempo no pior caso, que por simplicidade será chamado de complexidade.

Para tornar mais claras as idéias expostas, serão apresentados a seguir alguns exemplos.

Programação Dinâmica

A programação dinâmica é um método de desenvolvimento de algoritmos que consiste em, dado um problema, dividi-lo em subproblemas, solucionar os subproblemas, guardar os subresultados, combinar os subproblemas menores e subresultados para obter e resolver problemas maiores, até recompor e resolver o problema original.

Em [3] é apresentada uma especificação formal desse método, através de tipos abstratos de dados, cujo programa abstrato é reproduzido aqui.

```
Programa : PROG-DINÂMICA (p:P)
var: n,k, mínimo: ℕ; q:Q; m:M; r:R;
n ← tamanho (p);
q ← decompõe (p);
mínimo ← tamanho+ (q);
m ← inicializar (q)
para k de mínimo até n-1 faça
     (q,m) ← (combina, atualiza) (q,m)
fim-para
r ← recupera (m)
```

Esta especificação teve como uma das metas um estudo da complexidade associada ao método, o que facilitou o análise da complexidade, que é dada pela seguinte fórmula (sob algumas hipóteses simplificadoras)

$$T_{PD}(n) = T_{tamanho}(n) + T_{decompõe}(n) + T_{tamanho+}(n) + T_{inicializa}(n) + \sum_{k=\text{mínimo}}^{n-1} T_{(combina,\ atualiza)}(k) + T_{recupera}(n)$$

onde T_f é a complexidade de função f e T_{PD} a complexidade total do algoritmo.

Em [3] há um estudo mais detalhado dessa complexidade, com análise de casos e exemplos. Em [2] há uma outra especificação formal para a Programação Dinâmica.

Método Guloso

O Método Guloso é usado no projeto de algoritmos para problemas combinatoriais de omitização. O algoritmo assim projetado gera uma sequência de decisões baseadas em uma política de otimização local, de tal forma que nenhuma decisão é errônea.

O programa abstrato reproduzido aqui faz parte da especificação formal definida em [2] .

```
Programa  GULOSO (d:D)

var  r:R; a:A; i:I;
r ← init (d);
a ← choice (d);
enqto  finished  (a)  faça
       i ← select (a);
       a ← remove (i,a)
       se  feasible (i,r) então
           r ← alter (i,r)
       fim-se
fim-enqto
```

A complexidade é assim calculada:

$$T_{MG}(n) = T_{Inicialização}(n) + \sum_{i \in V} T_{Seleção}(n_i') + T_{Atualiza}(n_i'')$$

onde:

$$T_{Inicialização} = T_{init} + T_{choice}$$

$$T_{Seleção} = T_{finished} + T_{select} + T_{remove}$$

$$T_{Atualização} = T_{feasible} + j.T_{alter}$$

n_i' = tamanho (a) na i$\underline{\text{ésima}}$ iteração

n_i'' = tamanho (r) na i$\underline{\text{ésima}}$ iteração

$$j(m) = \bigvee_{(k,s) \in W_m} \text{feasible } (k,s)$$

$$W_m = \{(k,s) / K \in I \text{ e } s \in R, |s| = m\}$$

$$V = \{i / i \in \mathbb{N}\ i < \#\{b / b \in D \text{ e finished } (b) = 1\}\}$$

Aqui a maior dificuldade está na identificação dos casos em que o comando enqto será executado (cálculo de V) e na identificação dos casos em que o predicado do comando se é verdadeiro (cálculo de j). Sempre que for viável é aconselhável usar comandos do tipo para como no exemplo anterior, ao invés do comando enqto.

Convém lembrar que se está estudando a complexidade de tempo para o pior caso. No cálculo de W_m e V é feita uma análise localizada, é provável que alguns casos considerados nunca ocorram. Por exemplo, pode $(k,r) \in W_m$ mas não ser possível para qualquer entrada $d \in D$ o programa GULOSO atingir um estado em que a variável i tenha valor k,r valor s e a execução estar no comando se. Se não dificultar muito, no cálculo de V podem ser examinadas somente $b \in choice\ (D)$ ao invés de $b \in D$.

Divisão e Conquista

A Divisão e Conquista pode ser definida brevemente como segue. Dado uma instância de um problema, ela é decomposta em subinstâncias menores, que são resolvidas separadamente e então as soluções parciais são combinadas para se obter a solução da instância original. Se o tamanho ("volume" de dados) das subinstâncias é relativamente grande, com soluções não imediatas. O processo é aplicado novamente, para se obter subinstâncias menores, até que se obtenham subinstâncias tão pequenas que possam ser resolvidas fácilmente. A formalização da Divisão e Conquista apresentada por Veloso em [1] é

$$\text{Solução (d)} = \begin{cases} \text{direto (d)} & \text{se simples (d)} \\ \text{combine (solução (parte 1 (d)),..., solução (parte m (d)))} & \text{caso contrário} \end{cases}$$

E a complexidade é dada por

$$T_{\text{Solução}}(n) = \begin{cases} T_{\text{direto}}(n) & \text{se } n=1 \\ T_{\text{combine}}(n_1,\ldots,n_m) + \sum_{i=1}^{m} \text{parte } i\ (n) + \sum_{i=1}^{m} T_{\text{Solução}}(n_i) & \text{se } n>1 \end{cases}$$

onde n_i é o tamanho do dado parte i (n). Com pequena perda de generalidade, com intuito de simplificar o análise se supos que o tamanho do resultado é da mesma ordem de complexidade que o dado correspondente e que os dados simples são os de tamanho 1. (simples (d) $\Leftrightarrow$ d = 1).

Em [4] é feito um estudo bem extenso se casos, variando a complexidade das funções primitivas, m(o número de subproblemas) e o cálculo de n_i (regra que define a redução de tamanho dos problemas em cada nível de recursão).

A recursividade desse método dificulta o cálculo da complexidade de uma dada instância, já que para resolver a equação de recorrência de $T_{\text{Solução}}$ é necessário inicialmente encontrar uma candidata à complexidade e então provar por indução ser esta candidata a solução da equação.

Conclusão

O cálculo da complexidade de um programa abstrato antecipa o cálculo da complexidade de um algoritmo para a fase de projeto, facilitando a tarefa de melhorar o desempenho do mesmo. Pois, este cálculo evidencia quais as operações que mais pesam na complexidade.

Um estudo de casos como o realizado em [3] com a Programação Dinâmica e com a Divisão e Conquista em [4] auxiliam o projetista de algoritmo no desenvolvimento de algoritmos com bom desempenho.

A complexidade de classes de algoritmos paralelos também podem ser estudados dessa maneira. Em [5] é apresentado o estudo da complexidade de Divisão e Conquista para algoritmos paralelos, com os resultados comparados com a complexidade da versão sequencial.

Bibliografia

[1] - VELOSO, Paulo A.S "Divide-and-Conquer via data types" Analies VII Conf. Latinoamericana de Informática. Caracas, Venezuela 1980

[2] - WAGA, Cristina, F.E. "Métodos de resolução de problemas". Deptº de Informática da PUC/RJ. Dissertação de Mestrado. Rio 80p 1984

[3] - TOSCANI, Laira V. & VELOSO, Paulo A.S. Especificação Formal e Análise da Complexidade de Programação Dinâmica. RP49 CPGCC-UFRGS Porto Alegre, 50p 1980

[4] - TOSCANI, Laira V. & VELOSO, Paulo A.S. Divisão e Conquista: Análise da Complexidade. Anais do VI Congresso Brasileiro de Computação. Olinda, PE. p89-104 1986

[5] - TOSCANI, Laira V. & RIBEIRO, Celso C. Análise da Complexidade da Divisão e Conquista para Máquinas Paralelas com Estrutura de Árvore. (a ser publicado)

[6] - TOSCANI, Laira V. & RIBEIRO, Celso C. Análise da Complexidade de Algoritmos para Arquitetures Paraleles: Estudo da Técnica da Divisão e Conquista. Anais do 1º Simpósio Brasileiro de Arquitetura de Computadores. Processamento Paralelo. Gramado, RS p343-52 1987.