UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

ANAC – Uma Ferramenta para a Automatização da Análise da Complexidade de Algoritmos

por

MARCO ANTONIO DE CASTRO BARBOSA

Dissertação submetida à avaliação, como requisito parcial para
a obtenção do grau de Mestre em
Ciência da Computação

Prof.ª Dr.ª Laira Vieira Toscani
Orientadora

Prof.ª Dr.ª Leila Ribeiro
Co-Orientadora

Porto Alegre, dezembro de 2001

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Barbosa, Marco Antonio de Castro

ANAC – uma ferramenta para a automatização da análise da complexidade de algoritmos/por Marco Antonio de Castro Barbosa. – Porto Alegre: PPGC da UFRGS, 2001.

80f.: il.

Dissertação (mestrado) - Universidade Federal do Rio Grande do Sul, Programa de Pós-Graduação em Computação, Porto Alegre, 2001. Orientadora: Toscani, Laira Vieira. Co-orientadora: Ribeiro, Leila.

1. Algoritmos. 2. Complexidade de Algoritmos. 3. Análise Automática de Algoritmos. 4. Pior Caso. I. Toscani, Laira Vieira. II. Ribeiro, Leila. III. Título.

Agradecimentos

A realização deste trabalho certamente não foi uma tarefa fácil. Nada fácil é, também, escrever algumas palavras de agradecimento, pois há o receio de ser injusto e acabar esquecendo alguém. Tentei relacionar todos que direta ou indiretamente foram importantes na concretização desta conquista.

Vou começar por Deus, o patrão velho criador desta querência. Pelas grandes oportunidades que está pondo em meu caminho.

Meu agradecimento e meu muito obrigado à profa. Laira Vieira Toscani a quem aprendi a admirar e respeitar, não só como a brilhante educadora e pesquisadora que é, mas também como essa grande figura humana, com sua personalidade forte e com seu coração de *complexidade exponencial*.

Agradeço ao prof. e amigo Simão Sirineo Toscani, pois foi através de seu intermédio que conheci a profa. Laira e a área da Informática Teórica. Tive o privilégio de ser seu aluno e trabalhar ao seu lado como colega na Unicruz.

Embora não tenha auxiliado de forma direta na realização desta dissertação, não posso deixar de citar meu grande mestre e incentivador Marco Aurélio Spohn. Ele me incentivou a sempre tentar dar vôos mais altos.

Aos meus grandes amigos Audrei Rossato, Éder Contri e Giovani Librelotto. Colegas de faculdade e amigos pra vida toda. Com certeza estes podem ir muito mais longe do que suas pernas podem alcançar. Não há como não citar em especial ao Giovani pois este foi colega de faculdade, de mestrado, colega de apê, de trabalho na Unicruz e espero sinceramente que um dia possamos voltar a ser colegas novamente.

Como não citar o Grupo dos Semânticos ? Márcia, Luciana, Ronnie, Simone, Silvana, Bel e novamente o Giovani. Sem dúvida a melhor parte destes anos de mestrado foi a companhia de vocês. Sucesso a todos. E procurem não esquecer deste mau humorado aqui.

À Confraria do Intratáveis, esses são intratáveis só no nome porque são todos pessoas incríveis e admiráveis, meus agradecimentos ao Marilton, Vaneci, Liara, ao Marcos que ficou tão pouco, mas o suficiente para conquistar um amigo e meu agradecimento todo especial ao Carlos Morelli, que estava sempre pronto a ajudar e tirar dúvidas do seu programa ACME e a Aline Loreto por ter agüentado a choradeira este tempo todo, sempre pronta a ajudar.

Aos meus pais e irmãs, por estarem presentes nos bons e nos maus momentos.

A Luciana Rech minha amiga mais distante e ao mesmo tempo a mais presente.

A minha mais nova amiga Mádia, seu incentivo e apoio foram muito importantes nas horas que as “baterias” pareciam que iriam acabar.

Ao grande filósofo e pensador, Laurício pelos grandes “tratados filosóficos” que realizamos neste últimos meses, que sem dúvida auxiliaram a amenizar a ansiedade e o cansaço deste final de jornada.

A prof. Leila por seu apoio no forma de co-orientação deste trabalho.

Ao CNPq, pelo apoio na forma de bolsa, sem a qual a realização deste trabalho não seria possível.

E a todos aqueles que não foram citadas e que de alguma forma colaboraram na realização deste trabalho.

Sumário

Lista de Abreviaturas

ACE	Analyser Complexity Evaluator
ACME	Analisador de Complexidade Média
LUO	Lambda-Upsilon-Omega

Lista de Símbolos

$\forall$	quantificador universal (para todo)
$\in$	pertence
O	notação Big Oh
Ω	notação Big Omega
θ	limite assintótico exato

Lista de Figuras

Lista de Tabelas

Resumo

A análise de um algoritmo tem por finalidade melhorar, quando possível, seu desempenho e dar condições de poder optar pelo melhor, dentre os algoritmos existentes, para resolver o mesmo problema.

O cálculo da complexidade de algoritmos é muito dependente da classe dos algoritmos analisados. O cálculo depende da função tamanho e das operações fundamentais. Alguns aspectos do cálculo da complexidade, entretanto, não dependem do tipo de problema que o algoritmo resolve, mas somente das estruturas que o compõem, podendo, desta maneira, ser generalizados. Com base neste princípio, surgiu um método para o cálculo da complexidade de algoritmos no pior caso. Neste método foi definido que cada estrutura algorítmica possui uma equação de complexidade associada. Esse método propiciou a análise automática da complexidade de algoritmos.

A análise automática de algoritmos tem como principal objetivo tornar o processo de cálculo da complexidade mais acessível.

A união da metodologia para o pior caso, associada com a idéia da análise automática de programas, serviu de motivação para o desenvolvimento do protótipo de sistema ANAC, que é uma ferramenta para análise automática da complexidade de algoritmos não recursivos.

O objetivo deste trabalho é implementar esta metodologia de cálculo de complexidade de algoritmos no pior caso, com a utilização de técnicas de construção de compiladores para que este sistema possa analisar algoritmos gerando como resultado final a complexidade do algoritmo dada em ordens assintóticas.

Palavras-chave: algoritmos, complexidade de algoritmos, análise automática de algoritmos, pior caso.

TITLE: "ANAC – A TOOL FOR AUTOMATIZATION ANALISYS OF ALGORITHMS COMPLEXITY"

Abstract

The aim of algorithm analysis is to improve, whenever possible, its performance and to give conditions to choose the best option, among the existing algorithms to solve a given problem.

The calculation of algorithm complexity is very dependent upon the class of the analyzed algorithms. The calculation depends on the size function and on the fundamental operations. Some aspects of the calculation of the complexity, however, do not depend on the type of problem that the algorithm solves, but only on the structures that compose it, and thefore they can be generalized. Based upon this principle, a method for the calculation of algorithm complexity arose. In this method it was defined that each algorithmic structure had its corresponding complexity equation. This method propitiated the automatic analysis of algorithm complexity.

The main objective of automatic algorithm analysis is to make complexity calculation more accessible.

The union of the worst case methodology and the automatic analysis of programs served as motivation for the development of the prototype of system ANAC, which is a tool for automatic complexity analysis of non-recursive algorithms.

The objective of this work is to implement this methodology of algorithm complexity calculation in the worst case, using compiler construction techniques so that this system can analyze algorithms generating as output the algorithm complexity in assymptotic orders.

Keywords: algorithms, algorithm complexity, automatic algorithm analysis, worst case.

1 Introdução

Em seus estudos, datados do final do anos 60, Donald Knuth [KNU 68], argumentava sobre a importância do preparo de programas para um computador digital, segundo ele "o processo de preparar programas para um computador digital é essencialmente atrativo, não apenas porque ele pode ser economicamente e cientificamente recompensador, mas também porque ele pode ser uma experiência estética tal qual a composição de uma música ou de um poema".

Com uma visão um pouco mais prática, Garey & Johnson [GAR 79] em seu estudo sobre computadores, complexidade e intratabilidade, descreveram a crucial importância da construção de algoritmos, não apenas corretos, ou seja, que solucionem o problema, mas que também sejam eficientes, gerando o resultado em tempo de execução aceitável.

O conceito de algoritmo, juntamente com os métodos de desenvolvimento de algoritmos e técnicas de eficiência de algoritmos desempenham um papel fundamental e central em Ciência da Computação [AHO 74].

Muitos estudos sobre algoritmos tem tradicionalmente focalizado alguns problemas específicos, e os algoritmos para resolvê-los, sem se preocuparem em delinear as características estruturais comuns a certas classes de algoritmos. "Desenvolvimento de algoritmos: esta é a área da computação onde as pessoas argumentam sobre programas e, ao mesmo tempo, provam teoremas sobre programas, ao invés de simplesmente escreverem e depurarem programas" [TER 90]. Comentários deste tipo são feitos tanto por programadores de sistemas como por teóricos de computação. Existem ainda outros indivíduos que dizem: "desenvolvimento apropriado de algoritmos tem nos ajudado a economizar um montante financeiro considerável" [TER 90].

Papadimitriou afirmou em sua obra [PAP 94] que "Complexidade computacional é a área da ciência da computação que contempla a razões do porque alguns problemas são tão difíceis de serem resolvidos por computadores. Este campo virtualmente não existia na década de 70, tem sido tremendamente expandido e no início da década de 90 já compreendia a maior parte da atividade de pesquisa na ciência da computação teórica."

A complexidade de um algoritmo está relacionada com o esforço computacional necessário para a sua execução. Algoritmos podem ser avaliados por uma variedade de critérios de medidas. Os critérios mais freqüentemente utilizados são a taxa de crescimento do tempo ou espaço requerido para resolver grandes instâncias de um problema. Tais medidas são denominadas, respectivamente, de complexidade de tempo e complexidade de espaço. É denominado *tamanho* do problema, uma medida da quantidade de dados de entrada do problema. Por exemplo: o tamanho de um problema de multiplicação de matrizes pode ser a maior dimensão das matrizes a serem multiplicadas. O tamanho de um problema de grafo pode ser o número de nodos, ou número de arestas [TOS 01].

A análise da complexidade de algoritmos é expressa em função das operações fundamentais, as quais variam de acordo com o algoritmo, e cujo esforço de execução é função do volume de dados. As operações fundamentais são aquelas que, dentre as demais operações que compõe o algoritmo, expressam a quantidade de trabalho, portanto são extremamente dependentes do tamanho problema e do algoritmo.

A importância da construção de algoritmos eficientes pode ser observada em [KAR 86], onde são relatados os problemas do Caixeiro Viajante e dos Circuitos

Lógicos. Em [COO 83], são relatados os problemas: Transformada Rápida de Fourier (FFT), multiplicação de matrizes, identificação de números primos, dentre outros. Este exemplos expressam a necessidade que se tem de buscar uma solução satisfatória para problemas onde não se conheça algoritmos eficientes que os resolvam. Além da busca por algoritmos eficientes, outro fator importante em analisar a complexidade de algoritmos, é poder escolher, dentre diferentes algoritmos que resolvam o mesmo problema, com desempenho na execução também diferenciado, aquele que se sobressaia sobre os demais em termos de eficiência na execução.

O processo de análise de complexidade de algoritmos é uma tarefa que exige conhecimento e perícia do analista. É necessário que o usuário tenha conhecimento de complexidade de algoritmo e um bom conhecimento matemático. Isto tem motivado, ao longo dos anos, pesquisas com a finalidade de criar métodos e construir ferramentas que tornem o processo de análise algorítmica automático. A análise da complexidade de algoritmos é uma área da Ciência da Computação que nos últimos anos teve um crescimento significativo. No entanto, pouco esforço é destinado ao estudo da análise automática da complexidade de algoritmos.

A mais antiga tentativa de analisar automaticamente a complexidade de programas é atribuída a Wegbreit [WEG 75], que em 1975 desenvolveu o sistema METRIC. Le Métayer [MET 88] desenvolveu o sistema ACE utilizando uma abordagem similar a de Wegbreit. Wolf Zimmermann [ZIM 88] direcionou sua linha de pesquisa à análise no caso médio. Outros trabalhos significativos que surgiram foram os trabalhos de Kozen [KOZ 81], Ramshaw [RAM 79] e Hickey e Cohen [HIC 88]. Alguns destes importantes e relevantes estudos apresentavam algumas restrições no uso, tais como: dificuldade na análise dos resultados gerados, por gerarem equações bastante extensas e complicadas, ou ainda, dificuldade na utilizações de alguns sistemas, por necessitarem de linguagens pouco utilizadas para a descrição dos algoritmos (como é o caso dos Sistemas METRIC e ACE que utilizam a linguagem Lisp) ou por necessitarem de conhecimento de software matemáticos (com acontece no sistema Lambda-Upsilon-Omega que necessita do software matemático Maple) para obter a análise do algoritmo.

A análise da complexidade de um algoritmo é usualmente tratada de maneira muito particular, ou seja, é uma atividade muito dependente da classe dos algoritmos a serem analisados. No entanto, alguns conceitos são gerais, no sentido de que só dependem da estrutura do algoritmo, independente do problema que se está resolvendo, podendo ser desta forma generalizados. Esta idéia motivou o desenvolvimento de uma metodologia de cálculo de complexidade para estruturas algorítmicas para o pior caso [TOS 90].

A união da metodologia proposta por [TOS 90], associada com a idéia da análise automática de programas, serviu de motivação para o desenvolvimento do protótipo de sistema ANAC, que constitui-se em uma ferramenta para análise automática da complexidade de algoritmos não recursivos. Os objetivos deste sistema são:

- Fornecer um ambiente que guie o usuário nos passos necessários ao cálculo da complexidade de algoritmos;
- Servir como ferramenta de apoio ao ensino de complexidade de algoritmos, constituindo-se num ambiente de aplicação prática ao embasamento teórico adquirido no ensino de complexidade algorítmica;
- Calcular a complexidade de algoritmos no Pior Caso;
- Estimular o cálculo da complexidade de algoritmos, como um meio de tornar programas mais eficientes.

O propósito deste trabalho é a implementação da metodologia de [TOS 90] aliada com algumas técnicas de construção de compiladores para que o sistema possa analisar sintaticamente um algoritmo e, à medida que a análise sintática vá sendo executada e as estruturas algorítmicas vão sendo identificadas suas respectivas equações de complexidade vão sendo construídas. As equações obtidas são simplificadas e o resultado obtido é dado em ordem assintótica de complexidade.

A proposta da ferramenta ANAC difere dos demais trabalhos citados anteriormente pela simplicidade de seu uso, pela facilidade de escrita de seus programas por utilizar a linguagem Pascal-like e pelo resultado gerado em ordem assintótica de complexidade, o que não ocorre na maioria dos sistemas para análise automática que são difíceis de usar, de escrever algoritmos, ou ainda, de analisar os resultados.

Este trabalho está organizado da seguinte maneira: o capítulo 2 apresenta os conceitos de complexidade; o capítulo 3 descreve o protótipo do sistema ANAC; o capítulo 4 mostra o funcionamento do sistema; o capítulo 5 resume as principais ferramentas de análise automática de complexidade e faz um estudo comparativo com o sistema proposto neste trabalho; o capítulo 6 resume os principais resultados desse trabalho e sugere outros trabalhos na mesma linha. No anexo 1 é definida, através de uma BNF, a linguagem aceita pelo sistema ANAC. O anexo 2 apresenta mais alguns exemplos de aplicação do sistema ANAC e no anexo 3 estão os resumos dos artigos que foram publicados sobre este trabalho.

2 Análise da Complexidade Algorítmica

Por que analisar a complexidade de um algoritmo? Segundo Flajolet [SED 96], existem muitas respostas para esta questão, dependendo do contexto, algumas das possíveis respostas podem ser:

- A finalidade da utilização do algoritmo;
- A importância do algoritmo com relação a outros algoritmos que solucionem o mesmo problema, para que se possa decidir por qual deles optar na implementação da solução do problema;
- A dificuldade da precisão e análise da resposta requerida.

A mais simples razão para se analisar um algoritmo é para poder apurar suas características e avaliar a viabilidade de sua utilização prática, ou para poder comparar o algoritmo com outros algoritmos desenvolvidos com o mesmo objetivo do algoritmo em questão. É desejável saber quanto tempo uma implementação de um algoritmo particular irá consumir durante sua execução ou quanto espaço irá requerer, para poder certificar-se da eficiência do algoritmo.

Complexidade Computacional é a área da Ciência da Computação que elucida as razões do porquê alguns problemas são tão difíceis de serem resolvidos por computadores. Papadimitriou [PAP 94] considera a complexidade como "a intricada e esquisita interação entre computação (classes de complexidade) e aplicações (i. e., problemas)".

A análise de algoritmos é uma atividade que contribui para o entendimento fundamental da Ciência da Computação. Segundo Aho [AHO 82], a complexidade é o coração da computação.

Para Tarjan [TAR 87] o estudo de algoritmo é interessante e gratificante, não apenas pelas oportunidades que esta área oferece para afetar o mundo da computação, mas pela ricas e surpreendentes conexões entre problemas e soluções que esta área revela e pela oportunidade de compartilhar, de maneira criativa e estimulante, novas idéias e descobertas.

Nos últimos anos a pesquisa na área de algoritmos progrediu bastante. Esse processo foi alcançado com o desenvolvimento de algoritmos mais rápidos, tais como o da Transformada Rápida de Fourier – FFT [SED 96]. Mas, também houve, a importante descoberta de certos problemas para os quais todos algoritmos são ineficientes. Estes resultados tem despertado considerável interesse no estudo de algoritmos, e a área de projeto e análise de algoritmos vem tornando-se um campo de intenso interesse.

Questões levantadas por Aho [AHO 74]:

- Dado um problema, como pode se encontrar um algoritmo eficiente para sua solução ?
- Uma vez encontrado tal algoritmo, como pode-se comparar estes algoritmos com outros algoritmos que resolvam o mesmo problema ?
- Como pode-se julgar a qualidade de um algoritmo ?

Questões dessa natureza continuam sendo feitas por programadores e analistas preocupados com a eficiência do algoritmo.

2.1 Conceitos Básicos

A complexidade de um algoritmo está relacionada com o esforço computacional necessário para a sua execução.

Algoritmos podem ser avaliados por uma variedade de critérios de medidas. Os critérios mais freqüentemente utilizados são a taxa de crescimento do tempo ou espaço requerido para resolver grandes instâncias de um problema. Tais medidas são denominadas, respectivamente, de complexidade de tempo e complexidade de espaço.

É denominado *tamanho* do problema, uma medida da quantidade de dados de entrada do problema. Por exemplo: o tamanho de um problema de multiplicação de matrizes pode ser a maior dimensão das matrizes a serem multiplicadas. O tamanho de um problema de grafo pode ser o número de nodos, ou número de arestas.

A análise da complexidade de algoritmos é expressa em função das operações fundamentais, as quais variam de acordo com o algoritmo, e cujo esforço de execução é função do volume de dados. As operações fundamentais são aquelas que, dentre as demais operações que compõe o algoritmo, expressam a quantidade de trabalho, portanto são extremamente dependentes do tamanho problema e do algoritmo.

As principais medidas de complexidade de tempo são: complexidade no pior caso, complexidade no caso médio e complexidade no melhor caso, sendo que as duas primeiras são as de maior importância e interesse.

Neste trabalho a ênfase será dada à complexidade de tempo.

2.1.1 A Análise no Pior Caso

A complexidade no pior caso é geralmente a medida mais empregada na prática. Fixado um tamanho de entrada, a análise no pior caso é feita em relação ao número máximo de operações fundamentais necessárias para a resolução de qualquer problema do tamanho fixado, ou seja, a complexidade de tempo no pior caso (ou simplesmente complexidade) é um função $f(n)$ que é o máximo, sobre todas as entradas de tamanho n (i.é, o número de operações fundamentais executadas), o "tempo" total de execução do algoritmo.

Seu valor pode ser considerado como um limite de complexidade que não será ultrapassado, sendo portanto, uma garantia de qualidade mínima do algoritmo. O aspecto negativo da análise no pior caso é que leva em consideração os casos caóticos que dificilmente irão ocorrer na prática. A complexidade no caso médio, para determinados algoritmos, é mais realística do que a análise no pior caso.

2.1.2 A Análise no Caso Médio

Se a complexidade é tomada como uma complexidade "média" sobre todas as entradas do tamanho dado, então a complexidade é denominada de complexidade esperada ou complexidade média. A complexidade média de um algoritmo é usualmente mais difícil de se obter do que a complexidade no pior caso. A análise no caso médio é baseada nas distribuições probabilísticas dos dados de entrada do algoritmo. Para algoritmos utilizados freqüentemente é mais indicado calcular a sua complexidade média, pois a probabilidade de cair no pior caso pode ser mínima. A dificuldade no cálculo do caso médio está justamente nas distribuições probabilísticas dos dados de entrada, que nem sempre são conhecidas. Apesar disto, existem bons exemplos de aplicação deste cálculo, porém utilizando sempre distribuição uniforme. Um exemplo característico é o algoritmo de ordenação *Quicksort*, que apresenta desempenho médio $O(n \log n)$ [HOR 78], enquanto o seu desempenho no pior caso é $O(n^2)$, mas sabe-se que na prática o algoritmo *Quicksort* raramente terá este desempenho pessimista.

Algoritmos que não contenham nenhuma estrutura condicional terão as complexidades no pior caso e caso médio iguais, porque o algoritmo executará sempre o

mesmo número de operações para um mesmo tamanho de entrada, pois possui um só caminho de execução.

2.2 Definição de Complexidade

Esta seção apresenta algumas definições de complexidade retiradas de [TOS 90] e [TOS 01].

A complexidade de algoritmos consiste na quantidade de trabalho necessária para a sua execução, expressa em função das operações fundamentais, as quais variam de acordo com o algoritmo, e em função do volume de dados. As operações fundamentais são aquelas que, dentre as demais operações que compõe o algoritmo, expressam a quantidade de trabalho sendo, portanto, extremamente dependentes do problema e do algoritmo.

De acordo com Knuth [KNU 83], um *algoritmo* é um método abstrato para computar a solução de um problema. Um *problema*, segundo Veloso [VEL 81], é uma 3-upla p = <D, R, q>, onde D é o domínio das instâncias do problema, R é o domínio dos resultados e q é uma relação binária entre D e R que define o problema. Uma *solução* de p é uma função total $\alpha : D \rightarrow R$, tal que $(\forall d \in D)((d, \alpha(d)) \in q)$. Desta forma, 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 a o conjunto de todos os algoritmos que computam α. Para calcular a complexidade de um algoritmo $a \in \mathsf{a}$, deve-se determinar as operações fundamentais e definir a função tamanho do problema. Se houver mais de uma operação fundamental é necessário que se defina o peso de cada operação. Considere E o conjunto de todas as seqüências de execução das operações fundamentais. Obtém-se o diagrama mostrado na FIGURA 2.1:

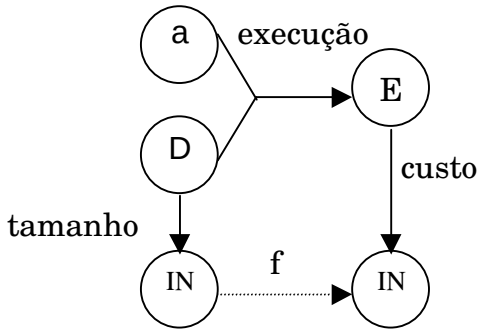


FIGURA 2.1 – Diagrama de Complexidade

onde:

- *execução* : a x D $\rightarrow$ E; *execução*(a, d) := seqüência de execuções de operações fundamentais efetuadas na execução do algoritmo a, com entrada d.
- *custo* : E $\rightarrow$ IN; *custo*(s) := comprimento da seqüência s, definido conforme o peso estabelecido para as operações fundamentais.
- *tamanho* : D $\rightarrow$ IN; *tamanho*(d) := tamanho da entrada d.

Determinadas as funções fundamentais e a função *tamanho*, o conjunto a é particionado (os algoritmos com mesmas funções fundamentais e mesma função *tamanho* ficam na mesma classe). A complexidade é então definida dentro de cada

classe. Considere A como uma dessas classes e defina a complexidade de cada elemento de A, como uma função de tipo IN $\rightarrow$ IN.

A *complexidade* é portanto um funcional de tipo A $\rightarrow$ (IN$\rightarrow$IN), isto é, *complexidade*: A $\rightarrow$ *f*, com *f* :={f/f: IN $\rightarrow$ IN}. Para a $\in$ A, *complexidade*(a) = f, f:IN $\rightarrow$ IN.

A função *tamanho* não é inversível. Dado n $\in$ N, para avaliar f(n) é necessário considerar todas as entradas d $\in$ D com *tamanho*(d):=n. É então calculado *custo*(*execução*(a,d)) e então, conforme o critério de complexidade desejado, é estabelecido f(n). Pode-se dizer que:

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

E *custo*(*execução*(a,d)) pode ser denominado desempenho de d em a, ou simplesmente desempenho de d, quando está claro quem é a, i.é.,

desempenho(d) := *custo*(*execução*(a,d)).

Dados a $\in$ a e n $\in$ IN, sejam

E_n := {d/d $\in$ D e *tamanho*(d) = n} e prob(d) a probabilidade de d ocorrer, então

avaliaçãoCPC(n) := max{*desempenho*(d) / d $\in$ E_n}

define a *Complexidade no Pior Caso*, e:

$$avaliaçãoCM\,(n) := \sum_{d \in E_n} prob(b) \cdot desempenho(d)$$

define a *Complexidade no Caso Médio*.

2.3 Ordens Assintóticas

Ao invés de calcular exatamente os tempos de execução em máquinas específicas, a maioria das análises conta apenas o número de operações fundamentais. Tal fato deriva do fato de que ao realizar uma medida empírica do tempo de execução de um algoritmo particular em um computador particular o resultado obtido fica estritamente ligado à linguagem de programação que foi utilizada na codificação do algoritmo e na máquina onde o programa foi executado. Desta forma, uma pequena mudança no programa poderia não causar mudança significativa no programa, mas poderia representar uma significativa mudança no tempo de execução do programa. Um fator importante na medida de complexidade é o crescimento assintótico da contagem de operações executadas. Por exemplo, num algoritmo para achar o elemento máximo entre *n* objetos, a operação fundamental seria a comparação dos objetos, e a complexidade seria o número de comparações efetuadas pelo algoritmo, para valores grandes de *n*.

A complexidade assintótica de tempo de computação de um algoritmo minimiza os efeitos de fatores que são dependentes da linguagem de programação e da máquina e concentra-se em determinar a ordem de magnitude da freqüência de execução das operações. O comportamento assintótico de um algoritmo é o mais procurado, já que para um volume grande de dados é que a complexidade torna-se mais importante. O algoritmo assintoticamente mais eficiente é melhor para todas entradas,

exceto entradas relativamente pequenas. Várias medidas de complexidade assintótica podem ser definidas, mas as mais usadas são 0, θ e Ω.[TOS 01]

A notação *0*: Esta notação define um limite assintótico superior, isto é

$$f(n) \text{ é } 0(g(n)) \text{ sss } (\exists c)\ (\exists n_0)\ (\forall n \geq n_0)\ f(n) \leq c \cdot g(n),$$

Isto é, a partir de um certo n_0, f(n) não cresce mais que $c \cdot g(n)$; ou f(n) tem o mesmo tipo de crescimento que g(n). Onde é um parâmetro que caracteriza a entrada e/ou a saída do algoritmo.

A notação θ: Esta notação define um limite assintótico exato, isto é

$$f(n) \text{ é } \theta(g(n)) \text{ sss } (\exists c)\ (\exists c')\ (\exists n_0)\ (\forall n \geq n_0)\ c \cdot g(n) \leq f(n) \leq c' \cdot g(n),$$

Isto é, existem constantes c, c' e n_0 tal que para todo valor de n maior ou igual a n_0, o valor de $f(n)$ está entre os produtos $c \cdot g(n)$ e $c' \cdot g(n)$.

A notação Ω: Esta notação define um limite assintótico inferior, isto é

$$f(n) \text{ é } \Omega(g(n)) \text{ sss } (\exists c)\ (\exists n_0)\ (\forall n \geq n_0)\ 0 \leq f(n) \leq c \cdot g(n),$$

Isto é, existem constantes c e n_0 tal que para valores de n iguais ou superiores a n_0, $f(n)$ é menor ou igual ao produto $c \cdot g(n)$.

Se $f(n)$ é $0(g(n))$, por abuso de linguagem também se escreve $f(n) = 0(g(n))$, ocorrendo o mesmo com as outras definições de ordem. Assim o cálculo da complexidade se concentra em determinar a ordem de magnitude do número de operações fundamentais na execução do algoritmo.

2.3.1 Relação entre as Ordens Assintóticas

A seguir é apresentada uma relação hierárquica, da menor para a maior, entre as ordens assintóticas.

O(1)	ordem constante
O(log n)	ordem logarítmica
O(n)	ordem linear
O(n log n)	ordem n log n
$O(n^2)$	ordem quadrática
$O(n^3)$	ordem cúbica
$O(2^n)$	ordem exponencial

Na FIGURA 2.2, pode-se observar graficamente como esta relação se apresenta. Verifica-se, através das curvas, o crescimento das funções relacionadas às ordens de complexidade. Para instâncias suficientemente grandes, é inegável a eficiência de um algoritmo cuja complexidade possua ordem logarítmica (log n), por outro lado, para estas mesmas instâncias, um algoritmo cuja complexidade seja dada por uma função de ordem exponencial (2^n) é totalmente ineficiente. Note também que a base do logaritmo é irrelevante nas ordens assintóticas, pois a troca de base se faz multiplicando por uma constante.

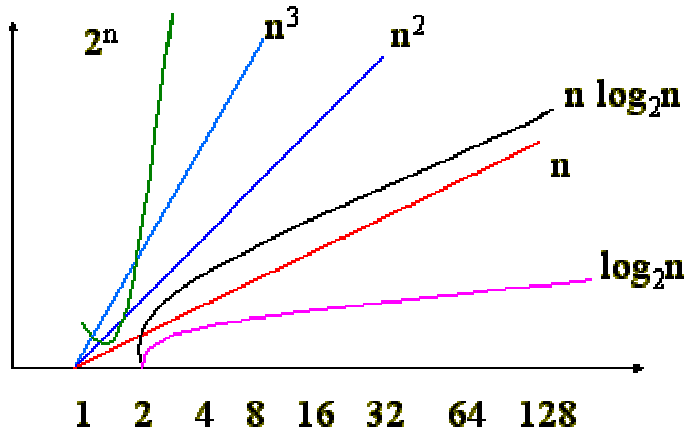


FIGURA 2.2 – Gráfico da relação entre as ordens assintóticas

A TABELA 2.1, apresenta números que exemplificam mais detalhadamente a relação entre as ordens assintóticas.

TABELA 2.1 – Ordem de Complexidade x Tamanho da Entrada

Função de Complexidade	**Tamanho do problema**			
	10	**10^2**	**10^3**	**10^4**
$log_2 n$	3.3	6.6	10	13.3
n	10	10^2	10^3	10^4
$n \log_2 n$	$0.33 * 10^2$	$0.17 * 10^3$	10^4	$1.3 * 10^5$
n^2	10^2	10^4	10^6	10^8
n^3	10^3	10^6	10^9	10^{12}
2^n	1024	$1.3 * 10^{30}$	2^{1000}	2^{10000}

No exemplo da TABELA 2.1, está sendo considerada uma relação dos tempos de execução de um algoritmo de uma ordem dada para entradas de tamanho dado. O desempenho do algoritmo que soluciona o problema será dado para quatro entradas de diferentes tamanhos, sendo elas: 10, 10^2, 10^3 e 10^4. Pode-se notar que nas primeiras linhas da Tabela a complexidade aumenta pouco em relação ao tamanho do problema, na última linha, porém, é assustador o aumento verificado. Através da Tabela pode-se avaliar que, o desempenho de um algoritmo cuja complexidade seja dada por uma função logarítmica é bastante superior aos demais, sendo que as funções polinomiais apresentam desempenho aceitáveis para grandes instâncias e algoritmos exponenciais apresentam desempenho totalmente insatisfatório, o que comprova sua ineficiência quando utilizados para instancias suficientemente grandes. Isto também justifica a

dicotomia que existe em complexidade na qual, algoritmos polinomiais são eficientes e algoritmos exponenciais ineficientes. E então os problemas: problemas com algoritmos polinomiais são tratáveis, problemas com algoritmos exponenciais são intratáveis.

2.4 Limites Superior e Inferior

Para um dado problema, pode-se ter vários algoritmos para resolvê-lo, cada um deles com complexidades diferentes. O *limite superior de complexidade de um problema* refere-se ao melhor algoritmo conhecido que o resolva.

Para saber se e quanto pode-se melhorar um algoritmo é preciso estabelecer um *limite inferior de complexidade de um problema* (que ele resolve) sobre a quantidade de recursos necessários, ou seja, qual a quantidade mínima necessária de recursos para a resolução do problema. Os problemas têm suas dificuldades, que não permitem o desenvolvimento de algoritmos melhores do que um certo limite de complexidade. O limite inferior de complexidade de um problema é portanto um resultado matemático.

Enquanto houver diferença entre os limites de complexidade inferior e superior para um problema, a questão pode progredir nos dois sentidos: um algoritmo melhor que o limite superior pode ser desenvolvido, ou um cálculo mais apertado de complexidade mínima pode ser alcançado, diminuindo a diferença entre esses dois limites. Ao desaparecer esta diferença a complexidade mínima do problema é alcançada.

Idealmente, os dois limites, inferior e superior, deveriam ser iguais, pois neste caso seria conhecida exatamente a quantidade de recursos que é tanto necessária quanto suficiente para resolver um problema.

Se um algoritmo utilizar exatamente esta quantidade de recursos, então, trata-se de um *algoritmo ótimo* para a tarefa, no sentido de que a quantidade de recursos utilizada por qualquer outro algoritmo para a tarefa será maior, ou no melhor caso igual a do algoritmo em questão.

A diferença entre o limite inferior e o superior fornece uma medida de quanto um algoritmo pode ser melhorado. Nem sempre, no entanto, é possível se construir algoritmos ótimos, nem calcular o limite inferior. Outra importante distinção entre limites inferior e superior é que o limite superior refere-se ao algoritmo, ao passo que o limite inferior é relacionado ao problema.

2.5 Complexidade das Estruturas Algorítmicas

O método de cálculo da complexidade de algoritmos, para o pior caso, com base na estruturas algorítmicas foi desenvolvido por [TOS 90]. Esta seção apresenta a definição completa deste método.

A análise da complexidade de algoritmos é uma atividade muito dependente da classe dos algoritmos a serem analisados. O cálculo depende essencialmente da função tamanho e das operações fundamentais. Este cálculo é feito de maneira muito particular a cada classe, sem a presença de uma sistemática que generalize o cálculo. Alguns aspectos do cálculo da complexidade, entretanto, não dependem do tipo de problema que o algoritmo resolve, mas somente das estruturas que o compõem, podendo, desta maneira, ser generalizados.

As principais estruturas algorítmicas em linguagens Pascal-like são:

Atribuição: **a** := **b ;**
Seqüência: **a** ; **b** ;
Condicional: if **a** then **b** else **c** ;
Iteração: for **k** = **i** to **j** do **a** ;
Iteração Condicional: while **a** do **b** ;

A descrição das metodologias utiliza as seguintes notações:

c(a) - Complexidade de a no pior caso
n - Tamanho de entrada
$t^k(n)$ - Tamanho de entrada após k-ésima iteração

2.5.1 Metodologia de Cálculo de Complexidade para o Pior Caso

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

A complexidade de **a** é "absorvida" pela complexidade de **b**, se e somente se

$$\exists c\ \exists N\ \forall n \geq N \text{ complexidade}(\mathbf{a})(n) \leq c \cdot \text{complexidade}(\mathbf{b})(n).$$

Neste caso, diz-se que:

$$\text{complexidade}(\mathbf{a}) + \text{complexidade}(\mathbf{b}) = \text{complexidade}(\mathbf{b})$$

Por exemplo, considere complexidade(**a**) e complexidade(**b**) polinômios, n^2 e n^3 respectivamente. Neste caso a complexidade de menor grau é absorvida pela de maior grau. Desta forma, complexidade(**a**) é absorvida por complexidade(**b**), ou seja, n^2 é absorvida por n^3.

2.5.1.1 Atribuição

A atribuição é apresentada na forma:

a := **b** ;

A complexidade associada a esta estrutura depende do tipo dos dados envolvidos.

$$c(\mathbf{a} := \mathbf{b}) = c(:=) + c(\mathbf{b})$$

A atribuição pode ser uma operação simples, como a atribuição de um valor a uma variável, ou uma atribuição mais complexa, como a inserção de um nodo num grafo, a atualização de uma matriz, dentre outros. A atribuição pode ainda requerer uma avaliação de **b**, que pode ser uma expressão ou uma função. Nestes casos, existe um esforço computacional associado a operação de atribuição c(:=) e um esforço associado a **b**.

2.5.1.2 Seqüência

Este estrutura tem a forma:

a ; **b** ;

A complexidade da seqüência é a soma das complexidades componentes, ou seja:

$$c(\mathbf{a}\ ;\ \mathbf{b})\ (n) = c(\mathbf{a})\ (n) + c(\mathbf{b})\ (t(n))$$

onde, $t(n)$ é o tamanho da entrada após a execução de **a**, dado que antes da execução era n.
A ordem de execução de **a** e **b** influi na complexidade resultante, pois ela pode variar se trocada sua ordem de execução. Geralmente, a complexidade de **a** é absorvida pela de **b**, ou vice-versa.

2.5.1.3 Condicional

A estrutura condicional pode apresentar-se de diversas formas, sendo a mais usual:

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

A complexidade desta estrutura é definida pela complexidade da avaliação da condição **a** mais a complexidade de **b** ou a complexidade de **c**, conforme o critério de complexidade a ser utilizado. Como esta se tratando de complexidade no pior caso, a complexidade é definida como a complexidade de **a** mais a complexidade máxima entre **b** e **c**. Ou seja;

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

Ocorre, porém, que *max* entre funções não tem uma interpretação padrão, e pode se observar que c(**b**) e c(**c**) não são simples naturais, mas sim funções de IN em IN.

Freqüentemente, a partir de certo ponto, c(**b**) fica sempre maior que c(**c**), ou vice-versa. Então, parece razoável tomar c(**b**) ou c(**c**) como *max*(c(**b**),c(**c**)), conforme o caso. Entretanto, pode acontecer, como no caso abaixo, que não exista essa dominância.

Para exemplificar essa situação, pode-se imaginar 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 dependente do número de nodos. Aumentando o número de arestas, aumenta somente a complexidade do ramo then e aumentando o número de nodos, somente aumenta a complexidade do ramo else. Desta forma, aumentando convenientemente a entrada, a complexidade de cada ramo pode superar a do outro. Nesse ponto, o máximo ponto a ponto pode ser usado, isto é, a função

$$max(c(\mathbf{b}),c(\mathbf{c}))(n) := max(c(\mathbf{b})(n),c(\mathbf{c})(n)).$$

Uma solução mais simplista, mas muitas vezes usada, é utilizar como máximo a soma ponto a ponto das duas funções:

$$(c(\mathbf{b})+c(\mathbf{c}))(n) := c(\mathbf{b})(n) + c(\mathbf{c})(n).$$

Na verdade, há várias possíveis escolhas para máximo assintótico de funções de naturais. Para definir o máximo entre funções, é preciso ter uma relação de ordem entre elas.

A estrutura condicional também pode apresentar-se num modo mais simples sem a presença do else:

if **a** then **b**

neste caso, a complexidade desta estrutura é simplificada:

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

2.5.1.4 Iteração

O caso mais simples de iteração não condicional (ou definida) é:

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

A execução da iteração causa a execução de **a** (j-i+1) vezes, com o valor de **k** variando de **i** até **j**. Considerando-se que as valores de **i** e **j** não são alterados na execução de **a**, o número de iterações é determinado e é (j-i+1). Pode ocorrer, entretanto, a situação onde o valor de **a** varia a cada iteração, por exemplo, alterando o tamanho da entrada, então tem que ser considerada a complexidade de cada iteração executada. Por estas razões, a complexidade desta estrutura tem dois casos a serem considerados:

Se o tamanho da entrada não varia com a execução de **a**, tem-se que:

c(for **k** = **i** to **j** do **a**) = (**j**-**i**+1).c(**a**)

Se a complexidade de da execução de **a** varia durante a iteração, tem-se que:

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

2.5.1.5 Iteração Condicional

As estruturas de iteração condicional (ou indefinida), podem assumir várias formas. A forma vista a seguir será o while, uma vez que o tratamento para as demais estruturas é similar.

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

Neste tipo de iteração **b** será executado sucessivamente enquanto a condição **a** for satisfeita.

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

2.6 Análise Automática de Algoritmos

A análise de algoritmos é uma área da Ciência da Computação que nos últimos anos teve um crescimento significativo. No entanto, pouco esforço é destinado ao

estudo da análise automática de algoritmos. Neste capítulo, procurou-se mostrar um breve histórico de alguns dos principais estudos realizados nesta área.

A mais antiga tentativa de analisar automaticamente a complexidade de programas é atribuída a Wegbreit [WEG 75], que em 1975 desenvolveu o sistema METRIC. Este sistema analisa especificações de programas e as converte em equações de recorrência.

Le Métayer [MET 88] desenvolveu o sistema ACE utilizando uma abordagem similar a de Wegbreit. O ACE analisa programas usando uma grande base de dados com regras que convertem especificações de programas em equações de complexidade.

Wolf Zimmermann [ZIM 88] direcionou sua linha de pesquisa à análise no caso médio.

Kozen [KOZ 81] desenvolveu uma visão semântica de programas probabilísticos.

Ramshaw [RAM 79], em sua tese, desenvolveu um framework lógico para a análise da complexidade de programas que é análoga ao sistema de Floyd-Hoare. A possibilidade de utilizar este framework para construir um analisador automático de desempenho é discutida em um artigo de Hickey e Cohen [HIC 88]. Seu sistema parece ter grande poder expressivo, mas há a necessidade de um complicado sistema de simplificações – ou até deduções – e existe a dificuldade de determinar quais classes de programas são suscetíveis ao tratamento automático.

3 O Protótipo ANAC

A análise de algoritmo tem por finalidade melhorar, quando possível, seu desempenho e, quando houver mais de um algoritmo para a resolução de um mesmo problema, poder optar pelo melhor dentre os algoritmos existentes. O processo de análise, no entanto, é uma tarefa que exige conhecimento e perícia do analista. Por conta disto, pesquisas tem sido realizadas com a finalidade de criar métodos e construir ferramentas que tornem o processo de análise de complexidade automático.

O processo automático, ou mesmo semi-automático, de análise algorítmica apresenta a possibilidade de se efetuar experimentos nos algoritmos e obter os resultados de forma quase que instantânea, por exemplo, alterando valores de laços de iteração, custos de funções, dentre outras instruções que podem aumentar ou diminuir o tempo de computação de um algoritmo.

No ensino de Projeto e Análise de Algoritmos a utilização de uma ferramenta de análise automática de complexidade é motivadora do desenvolvimento de algoritmos eficientes e uso de técnicas que visam melhorar o desempenho do algoritmo.

Neste trabalho foram apresentadas, algumas pesquisas relacionadas ao tema da mecanização do processo de análise algorítmica, com a descrição de alguns sistemas e de metodologias para a automatização da análise de algoritmos.

Foi motivado nestas idéias que teve início o desenvolvimento do protótipo do sistema denominado ANAC – Analisador de Complexidade.

3.1 Motivação da Construção do ANAC

A base para o desenvolvimento do sistema ANAC é a metodologia de cálculo de complexidade para estruturas algorítmicas proposta em [TOS 90]. Esta metodologias englobam as principais estruturas que podem estar presentes em um algoritmo, por exemplo: atribuição, seqüência, condicional, iteração e iteração condicional. Esta metodologia é direcionada à análise no pior caso.

O sistema ACME – Analisador de Complexidade Média [SIL 98], foi desenvolvido com o propósito de adaptar a metodologia de [ROS 97] para uma ferramenta que automatizasse o processo de cálculo de complexidade de algoritmos no caso médio e também serviu de inspiração para o desenvolvimento do sistema ANAC.

3.2 Objetivos do Sistema

O sistema ANAC é uma ferramenta de apoio ao cálculo de Complexidade de Algoritmos. Os principais objetivos deste sistema são:

- Dar suporte ao desenvolvimento de algoritmos eficientes;
- Servir como ferramenta de apoio ao ensino de complexidade de algoritmos, constituindo-se num ambiente de aplicação prática ao embasamento teórico adquirido no ensino de complexidade algorítmica;
- Calcular a complexidade de algoritmos no Pior Caso;
- Estimular o cálculo da complexidade de algoritmos.

O sistema analisa algoritmos não recursivos escritos na linguagem Pascal-like englobando as principais estruturas presentes em linguagens imperativas: atribuição, seqüência, condicional, iteração e iteração condicional, calculando de forma semi-automática a equação de complexidade e resolvendo-a sempre que forem fornecidos os

dados necessários para tal operação. O algoritmo pode conter chamadas a procedimentos não especificados, cuja complexidade deve ser fornecida pelo usuário.

O sistema foi implementado na linguagem de programação Java, pela portabilidade desta linguagem e pelo propósito de disponibilizar o uso do sistema ANAC através da Internet.

3.3 Etapas da Construção do Protótipo ANAC

A primeira etapa da construção do protótipo ANAC foi a definição da linguagem a ser suportada por ele. A linguagem escolhida foi um Pascal-like, por estar mais próxima da linguagem algorítmica que é amplamente utilizada pelo meio acadêmico, que é o perfil do usuário a quem o sistema se destina. A descrição completa da linguagem de expressão dos algoritmos encontra-se no Anexo 1.

A segunda etapa foi a definição e construção de um analisador léxico-semântico. Num primeiro momento pensou-se em utilizar a ferramenta Jack [MCM 99], como foi originalmente denominada, ou JavaCC, como é conhecida atualmente. Esta ferramenta é um sistema que gera de forma automática o analisador léxico-semântico, cujo princípio de funcionamento é o mesmo dos sistemas Lexx-Yacc encontrados nos ambientes Unix. A utilização do JavaCC foi descartada por ser um sistema relativamente novo. O desenvolvedor do sistema, Chuck McManis [MCM 99], considerava o JavaCC, que encontra-se em sua versão 0.5, o "meio do caminho" da construção completa e definitiva do sistema e solicitava que os desenvolvedores Java o auxiliam-se na detecção de *bugs* e no desenvolvimento de novas aplicações com o sistema. Isso gerou certo receio em utilizar este sistema no desenvolvimento da gramática da linguagem Pascal-like para o sistema ANAC. Outra razão é o controle maior que se teria ao desenvolver um analisador léxico-semântico próprio. Com a linguagem Pascal-like definida e sua correspondente BNF (Anexo 1), o passo seguinte foi a definição de qual método e estratégia de análise seria utilizada no analisador léxico-semântico. Para o desenvolvimento do sistema ANAC, entendeu-se que uma estratégia *top-down* é a mais indicada por estar próxima do funcionamento da metodologia utilizada. Para implementar a estratégia *top-down* foi escolhido o método de análise preditiva-tabular [PRI 00], por ter sido julgado o de mais simples implementação. O analisador foi implementado na linguagem de programação Java.

A terceira-etapa foi a adaptação do analisador semântico para que este gerasse a equação de complexidade dos algoritmos. Esta foi a etapa mais difícil da implementação do sistema. Como se pode ver, nos livros de Análise de Algoritmos [AHO 74], [HOR 78], [COR 90], [KNU 68] e [TER 90] a análise da complexidade de um algoritmo, é tratada de maneira muito particular, ou seja, é uma atividade muito dependente da classe dos algoritmos a serem analisados, entretanto, segundo [TOS 90], alguns aspectos do cálculo da complexidade não dependem do tipo de problema que o algoritmo resolve, dependem das estruturas que compõem o algoritmo, podendo ser desta forma generalizados. Um analista que utiliza esta metodologia e faz a análise de maneira manual pode verificar a generalidade deste método. No entanto, por mais genérico que seja esta metodologia, toda estrutura algorítmica possui alguma particularidade, que no processo manual é, de certa forma, simples de verificar e tratar, já para um processo computacional identificar tais particularidades não é tarefa tão trivial e tratá-la pode ser tarefa ainda mais trabalhosa.

A seguir serão vistas as estruturas algorítmicas, suas particularidades e o tratamento que foi empregado na implementação destas estruturas no sistema ANAC.

Definição e Chamada de Procedimentos: O sistema ANAC permite que o

usuário crie procedimentos sem a necessidade de especificar seu corpo de comandos. Para tanto, o usuário deve definir o nome do procedimento e especificar a complexidade relativa à avaliação deste procedimento. Estas informações são inseridas na tabela de procedimentos que será armazenada pelo sistema na forma de arquivo, para que possa ser reutilizada posteriormente.

Estrutura de Atribuição: Conforme pode ser observado na seção 2.5 a atribuição pode ser uma operação simples, como a atribuição de um valor a uma variável, ou mais complexa, como a atribuição do valor de uma função. Quando a atribuição é simples, o esforço computacional associado a esta estrutura é tido como constante (1), quando complexa, a definição da complexidade irá depender da expressão ou da função.
Como foi visto anteriormente, o usuário pode usar procedimentos pré-estabelecidos e não definidos, ou ainda, pode criar e definir novos procedimento tipo: "ordene a lista". Durante o processo de análise, caso o usuário tenha optado por utilizar a tabela de procedimentos, quando o sistema encontrar no algoritmo uma atribuição, a tabela de procedimentos é percorrida, se houver na atribuição algum destes procedimentos, o sistema tem informações suficientes para construir a equação de complexidade correspondente. Se, por outro lado, não houver nesta atribuição nenhum comando ou procedimento, que esteja armazenado nesta tabela, o sistema assume como uma atribuição simples, cuja complexidade é constante.

Estrutura Condicional: A forma mais comum de uma estrutura condicional é: if <*condição*> then S_1 else S_2, ou, if <*condição*> then S_1. Para calcular a complexidade da estrutura condicional no segundo caso, é necessário apenas somar a complexidade da avaliação da condição com a complexidade do ramo then, nesta segunda forma não existe o ramo else. Para calcular a complexidade da estrutura condicional, com os ramos then e else, é necessária uma avaliação da condição e a determinação do valor de complexidade "máximo" dos ramos then e else.. O significado de "máximo" nesse contexto foi bem explorado em [TOS 01] e na seção 2.2. Como foi referenciado em [TOS 01], uma interpretação válida para "máximo" numa avaliação assintótica é a soma, que foi adotada aqui. Como já foi dito anteriormente, no sistema ANAC, a equação de complexidade é construída de maneira *top-down*. Por isso, o sistema irá identificar o comando if e deveria, neste momento, saber se o comando é simples, ou seja, existe apenas o ramo then, ou composto, com os ramos then e else. Foi implementado uma estrutura única, ou seja, com a complexidade considerada como a soma da complexidade da condição mais a complexidade do bloco de comandos limitados pelos comandos if-then e endif. Neste caso, o comando poderia ou não apresentar o ramo else e a equação de complexidade correspondente estaria sempre corretamente definida.

Estrutura de Iteração Incondicional: A estrutura de iteração considerada no sistema ANAC é a do tipo for k = i to j do **a**. A execução da iteração causa a execução de **a** (j-i+1) vezes, com o valor de k variando de i até j. Considerando-se que os valores de i e j não são alterados na execução de **a**, o número de iterações é determinado e é (j-i+1). Pode ocorrer, entretanto, a situação onde o valor de **a** varia a cada iteração, por exemplo alterando o tamanho da entrada, então tem que ser considerada a complexidade de cada iteração executada. Por estas razões, a complexidade desta estrutura tem dois casos a ser considerados, quando o tamanho da entrada não varia com a execução de **a** e quando a complexidade da execução de a varia durante a iteração. Pode-se observar,

nesta estrutura, que existem diferentes formas de iteração a primeira é um caso particular da segunda, portanto nesta estrutura implementou-se a idéia de uma estrutura cujo tamanho da entrada esteja variando, e neste caso, a equação de complexidade é dada na forma de um somatório das partes que serão iteradas pelo comando for. Exemplo:

```
for i:= 1 to n do
   a := b;
endfor;
```

No algoritmo o comando for irá fazer a iteração da atribuição *a := b* exatamente (n-1+1) vezes, ou seja *n* vezes. A estrutura *a := b* não sofre variação de complexidade a cada iteração, porém, no sistema ANAC a complexidade de tal algoritmo seria dada por:

$$\text{c (for1)} = \sum_{i=1}^{n} c(a := b) \text{ ou c (for1)} = \sum_{i=1}^{n} (1)$$

Neste caso, pode-se observar que a complexidade de um somatório de *n* vezes um valor constante vai ser definido pelo limite superior do somatório, ou seja, a complexidade desta estrutura será O(n), pois tal complexidade seria dada pela soma de 1 + 1 +...+ 1, *n* vezes, o que resultaria em *n*.1 = O(n).

Estrutura de Iteração Condicional: A estrutura de iteração condicional (ou iteração indefinida), pode assumir várias formas. A forma vista a seguir será o while, uma vez que o tratamento para as demais estruturas é similar. Neste tipo de estrutura a iteração será executada sucessivamente enquanto uma condição for satisfeita. A complexidade deste tipo de estrutura está associada ao número de vezes que a iteração ocorrer. A diferença entre este tipo de iteração e a iteração não condicional, ou definida, é que na iteração não condicional se consegue precisar o número de iterações que serão executadas, por outro lado, na iteração condicional o número de iterações geralmente não fica determinado no início da execução. Este número é definido pela função booleana que limita a execução e portanto não se conhece a *priori*. É necessário estimar este valor.

Exemplo: considere os seguintes algoritmos:

```
Program whi1;
Begin
  i := 0;
  While (i ≤ n) do
     A[i] := A[i+1];
      i := i + 1;
  Endwhile;
end.
```

(a)

```
program whi2;
begin
   while (nro >= 0 ) do
      read(nro);
   endwhile;
end.
```

(b)

No algoritmo (a), o comportamento da estrutura de iteração condicional (indefinida) é semelhante ao da iteração não condicional (definida). Neste exemplo, a variável i irá variar de 1 até *n*. Desta forma pode-se precisar o número exato de iterações que serão executadas. No algoritmo (b), não se pode precisar o número iterações que serão executadas.

No sistema ANAC toda iteração será assumida com definida. Desta forma quando o sistema encontrar uma iteração while irá solicitar que o usuário indique um limite superior referente ao número de vezes que a iteração será executada.

A última etapa da construção do sistema ANAC foi o desenvolvimento do procedimento de simplificação da equação de complexidade gerada pelo sistema. Tal procedimento é definido mais detalhadamente a seguir.

Processo de Resolução das Equações de Complexidade no sistema ANAC:

O sistema ANAC implementa a metodologia descrita em [TOS 90] apresentada no capítulo 2, através de uma análise simbólica *top-down*. Quando uma estrutura algorítmica é identificada a equação de complexidade referente a ela é definida. As complexidades das partes são identificadas por variáveis que são detalhadas (calculadas por outras equações de complexidade) à medida que a estrutura algorítmica correspondente é identificada.

À medida que o algoritmo vai sendo analisado e suas estruturas são identificadas é construída uma árvore-sintática correspondente. Nesta árvore a raiz é a complexidade do algoritmo, que é constituído de partes que podem ser: atribuições, estruturas condicionais, iterações definidas ou indefinidas, ou seqüências. Uma estrutura por sua vez é também composta por partes cada uma gerando um ou mais filhos.

A seguir será detalhado este processo.

A operação fundamental é a atribuição a qual é associado o custo 1. A complexidade dos testes é requerida ao usuário e complexidade dos delimitadores de escopo (end, endif, endfor e endwhile) é atribuído o valor zero.

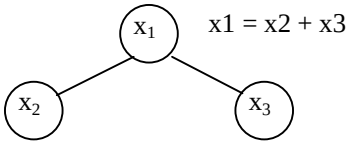
A seguir será detalhado o processo de análise e identificação das estruturas algorítmicas. Para tal descrição será usada a seguinte notação:

S_i é uma estrutura do algoritmo.
x_i = complexidade da estrutura S_i;
t(n) = tamanho da entrada

O algoritmo é considerado sempre, em princípio, como uma seqüência. É identificada a primeira estrutura e a continuação é atribuída a uma variável.

Seqüência: c(a;b)(n) = c(a) (n) + c(b) (t(n))

TABELA 3.1 – Seqüência

Estrutura	Análise do programa	Árvore da Seqüência
S_2 ; S_3 ;	x1 = x2 + x3	

O cálculo da complexidade de seqüência, constitui-se na soma das complexidades das partes. A TABELA 3.1, ilustra uma representação da complexidade da seqüência, onde x_1 representa a soma das partes S_1, S_2, cujas complexidades estarão definidas em x_1, x_2.

A seqüência é considerada a estrutura básica.

Condicional: if <*condição*> then S_2 else S_3

Considerando x1 = c (if <*condição*> then S_2 else S_3) ou seja, x1 é igual a complexidade da estrutura:

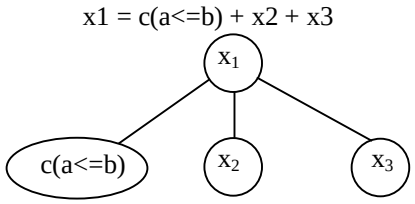
(if <*condição*> then S_2 else S_3).

Para se obter a complexidade desta estrutura as complexidades da condição e das estruturas S_2 e S_3 devem ser avaliadas. A variável x3, é utilizada para representar a complexidade do programa que segue. Para exemplificar esse processo a TABELA 3.2 mostra a estrutura condicional, a análise da complexidade dessa estrutura no pior caso, pelo sistema ANAC e a ilustração da árvore criada para a estrutura condicional.
O sistema ANAC, no cálculo da complexidade da estrutura condicional usa a seguinte equação:

c(if <*condição*> then S_2 else S_3) = c(*condição*) + c(S_2) + c(S_3)

TABELA 3.2 – Estrutura Condicional

Programa	**Análise do programa**	**Árvore da Estrutura Condicional**

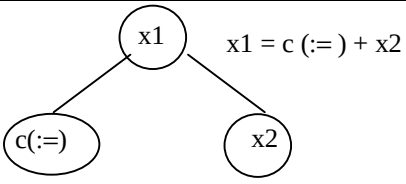
if a <= b then
S_2;
else
S_3;
endif;
	x1 = c (a<=b) + x2 + x3
	x2 = c (S_2) + x4
	x4 = c (S_3) + x5
	x5 = c (endif) = 0
		

O sistema irá construir a árvore com a raiz x1 e os dois filhos x2 e x3. Para a construção da equação o sistema irá percorrer o ramo correspondente a x2 até que encontre um nodo folha, ou seja, que não possua mais filhos e que o seu valor correspondente possa ser definido. Ao ocorrer essa situação o sistema retorna na árvore atribuindo o valor calculado à variável de complexidade correspondente a esta estrutura, no nodo pai. No nodo pai duas situações podem ocorrer: 1) não haver mais filhos naquele nodo pai, neste caso já existe informação suficiente para o valor da complexidade ser definido. 2) o pai pode ter mais filhos, neste caso o sistema irá percorrer o caminho definido pelo próximo filho até que encontre um nodo folha e possa retornar na árvore e ter, desta forma, subsídios para a avaliação da equação de complexidade correspondente. Isto é, a árvore é percorrida em profundidade, da esquerda para a direita.

Atribuição: c **(a := b**) = c (**:=**) + c (**b**)

A avaliação desta estrutura pode ser bastante simples, como o caso de uma atribuição de um valor a uma variável, ou pode ser necessário a avaliação de uma expressão ou função.

TABELA 3.3 – Estrutura de Atribuição

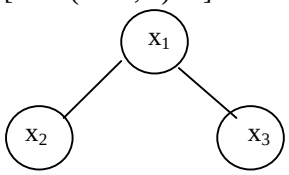
Programa	**Análise do programa**	**Árvore da Estrutura Atribuição**
a := b; S_2;	x1 = c (:=) + x2	

A complexidade do algoritmo da TABELA 3.3 será a complexidade da estrutura de atribuição c(:=) mais a complexidade da avaliação de b, mais a complexidade da próxima estrutura S_2 ainda não foi identificada e que será representada por x2.

Iteração Incondicional: c(for **k** = **i** to **j** do **a**)(n) = $\sum_{k=0}^{j-i} c(\boldsymbol{a})(t^k(n))$

TABELA 3.4 – Estrutura de Iteração

Programa	**Análise do programa**	**Árvore da Iteração**

for k := 1 to n do
S_2;
endfor;
S_3;
	x1 = c (for k:= 1 to n do S2) + x3
	x1 = [SUM(k=1 , n) x2] + x3
	x2 = c (S_1) + x4 = 1 + x4
	x4 = c (endfor) = 0
		x1 = [SUM(k=1 , n) x2] + x3
		 

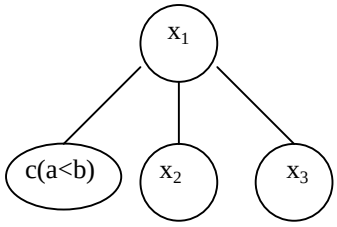
No exemplo da TABELA 3.4, a primeira estrutura encontrada foi a iteração, logo, a complexidade deste algoritmo é o somatório da complexidade da estrutura S_2 para cada vez que ela for executada mais a complexidade de uma próxima estrutura (S_3), identificada por x3. Para se obter x1, o sistema irá percorrer a árvore até conseguir obter a definição de x2 e seus filhos e irá retornar para obter x3, desta forma o sistema consegue obter x1.

Iteração Condicional: while a do b ;

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

TABELA 3.5 – Estrutura de Iteração Condicional

Programa	**Análise do programa**	**Árvore da Iteração Condicional**

while a < b do
S_2;
endwhile;
S_3;
	x1 = c(while a < b do S2) + x3 =
	x1 = c (a<b) + [SUM (i = 1 , n) x2] + x3
	x2 = c (S2) + x4 = 1 + x4
	x4 = c (endwhile) = 0
		x1 = c(a<b) +
		[SUM(i =1,n) +x2] + x3
		 

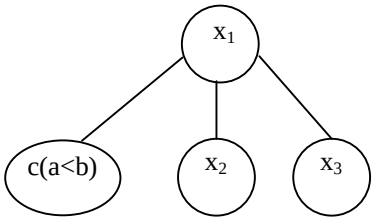
Como pode ser visto na TABELA 3.5, a complexidade do algoritmo, x1, é definida por c(a < b), mais o somatório da estrutura S_2, referente ao número de iterações que serão executadas. Exceto pela complexidade da avaliação da condição, o comportamento restante do cálculo da complexidade da iteração condicional, será igual ao cálculo da iteração não condicional.

Exemplo de Aplicação:

A TABELA 3.6 ilustra o exemplo de um algoritmo.

TABELA 3.6 – Exemplo de Aplicação

Programa	**Análise do programa**	**Árvore da Seqüência**

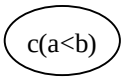
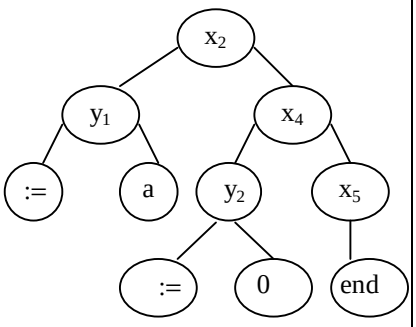
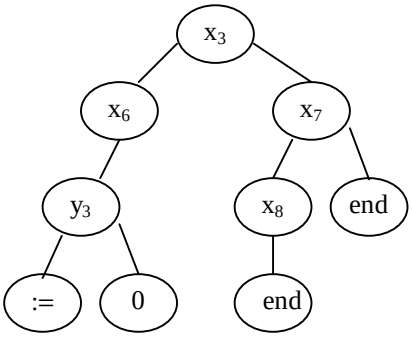
if a < b then
 	
vet[i] := a;
else
vet[i] := b;
endif;
for i:=1 to n do
vet[i] := 0;
 	
endfor;
end.
	X1 = c(a<b) + x2 + x3 =
	X1 = 1 + x2 + x3
	X2 = c(vet[i] := a) + x4 = 1 + x4
	X4 = c(vet[i] := b) + x5 = 1 + x5
	X5 = c (endif) = 0
	X4 = 1 + 0 = 1
	X2 = 1 + 1 = 2
	X3 = c (for i:= 1 to n do x6) + x7
	x3 = [SUM(i=1 , n) x6] + x7
	X6 = c(vet[i] := 0) + x8 = 1 + x8
	X8 = c (endfor) = 0
	X6 = 1 + 0 = 1
	X7 = c (end) = 0
	X3 = [SUM(i=1 , n) 1] + 0 =
	X3 = n + 0 = n
	X1 = 1 + 2 + n = n
	O (n)
		x1 = c(a<b) + x2 + x3
		 

(a)	(b)	(c)

Neste exemplo a primeira estrutura encontrada pelo sistema ANAC é a estrutura condicional, logo, a complexidade deste algoritmo será obtida pela soma da complexidade da avaliação da condição, mais o restante da estrutura condicional, que são as estruturas existentes no seu bloco de comandos (S_2), mais a próxima estrutura S_3. A coluna (a) da TABELA 3.7 representa a complexidade da condição da estrutura if. No momento em que o sistema identifica a estrutura if ele irá solicitar que o usuário informe a complexidade da avaliação desta condição. Caso o usuário não informe esta complexidade o sistema irá assumir a complexidade constante (1), que foi a complexidade da avaliação definida para este exemplo. As variáveis x2 e x3 irão armazenar as complexidade de S_2, que são as estruturas existentes no bloco de comando delimitados e pelos comando if-then e endif e S_3 que será a estrutura de iteração for. A coluna (b) da TABELA 3.7 representa o ramo gerado para a variável x2. A complexidade da variável x2 será dada pela soma da primeira estrutura encontrada dentro do bloco if-then, que é uma estrutura de atribuição *vet[i] := a*, mais uma próxima estrutura ainda não identificada. Para a definição da complexidade da atribuição *vet[i]:= a* o sistema irá utilizar uma variável auxiliar implícita que não aparece no processo mostrado ao usuário. A variável y1 será usada para armazenar a complexidade da atribuição, que é a soma do custo da operação de atribuição (:=), mais a complexidade da avaliação de **a**. Neste momento o sistema irá verificar se a expressão **a** contém procedimentos definidos pelo usuário. No caso de haver procedimentos o sistema irá buscar na tabela o custo relativo a este procedimento e este valor será utilizado no cálculo da atribuição. Esta atribuição representa um nodo folha, e o sistema irá prosseguir encontrando a próxima estrutura que foi identificada por x4. A complexidade de x4 é dada por uma estrutura de atribuição, mais uma próxima estrutura

identificada por x5. Para o cálculo da estrutura de atribuição a variável y2 é utilizada e o processo de cálculo segue a mesma metodologia citada anteriormente. Após obter a complexidade desta atribuição o sistema prossegue e irá encontrar a estrutura cuja complexidade é x5 endif, que é uma estrutura que indica um fim de bloco e que terá custo zero (0). Após esta etapa a complexidade da variável x2 estará definida, falta ainda definir a variável x3 para que a complexidade do algoritmo possa ser definida. A complexidade x3 é dada pela soma da complexidade da estrutura de iteração for e uma continuação cuja complexidade é representada pela variável x7. A complexidade da estrutura for será obtida através de informações do usuário, na forma de somatório. No momento em que esta estrutura for identifica será solicitado que o usuário informe a variável, o limite inferior e limite superior do somatório. Neste exemplo foi definido um somatório variando de i = 1 até *n*. Desta forma, a complexidade do for será obtida através da soma de *n* vezes a complexidade do for x6 ou da estrutura S_6, que está inserida no bloco da estrutura for, mais a próxima estrutura ainda não identificada de complexidade x8. A estrutura S_6 é identificada como uma atribuição. Para o cálculo de sua complexidade a variável auxiliar y3 será utilizada. Após definir a complexidade desta atribuição o sistema irá encontrar a estrutura S_8 que é a estrutura de fim de bloco endfor, cuja complexidade é zero (0). Por fim, o sistema irá encontrar a estrutura S_7 end, que terá custo zero (0). Uma vez definido seu valor de complexidade o sistema pode retornar na árvore e realizar a soma destas estruturas para obter o valor de x1 que irá representar a complexidade do algoritmo.

TABELA 3.7 – Ramos do Exemplo de Aplicação.

(a)	(b)	(c)
		

4 A Utilização do Sistema ANAC

O sistema apresenta-se ao usuário como uma ferramenta para edição de textos bastante simples.

O sistema possui uma janela com menu de opções **Arquivo**, **Editar**, **Tabela**, **Analisar** e **Ajuda**. O processo de análise tem inicio quando o usuário aciona o sub-item **Iniciar** no menu **Analisar**.

Primeira Etapa:

O sistema pode analisar algoritmos prontos e carregados através da opção do menu **Arquivo/Abrir**, ou o algoritmo pode ser escrito na área de texto do sistema utilizando-se a opção do menu **Arquivo/Novo**. Estas opções estão representadas na FIGURA 4.1. Além destas opções o Menu **Arquivo** possui as opções **Salvar**, que irá fazer a atualização de um arquivo carregado, sobregravando o arquivo com o mesmo nome e no mesmo diretório e unidade de disco. A opção **Salvar Como** irá abrir uma janela que permitirá ao usuário modificar o nome, diretório e unidade de disco do arquivo a ser gravado. A opção **Sair** irá encerrar a execução do sistema ANAC.

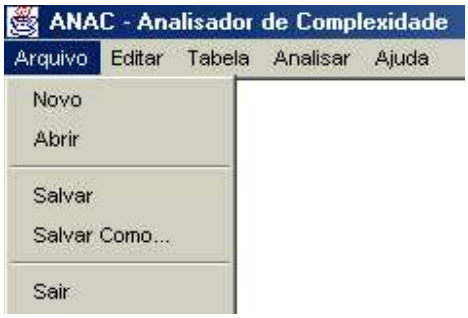


FIGURA 4.1 – O Menu Arquivo

Segunda Etapa:

Uma das principais características do sistema ANAC é a possibilidade de o usuário definir funções ou utilizar funções pré-estabelecidas. Na FIGURA 4.2, pode-se observar o Menu Tabela com as opções de **Utilizar Tabela**, **Visualizar Tabela**, **Alterar Tabela**, **Inserir Item** e **Excluir Item**, que são as opções para utilizar as funções já existentes ou criar novas.

FIGURA 4.2 – O Menu Tabela

A opção **Utilizar Tabela**, quando selecionada pelo usuário, irá carregar e disponibilizar a tabela para uso do sistema. Nesta tabela estarão definidos os procedimentos usados pelo usuário e a complexidade de cada procedimento. Durante o processo de análise do algoritmo, ao encontrar um procedimento externo, o sistema irá buscar na tabela a complexidade daquele procedimento e o tamanho da entrada e desta forma terá subsídio para realizar o cálculo da complexidade do algoritmo. A opção **Visualizar Tabela** permite a visualização da tabela, com o nome do procedimento, o tamanho da entrada e a complexidade. Um exemplo pode ser observado na FIGURA 4.3, onde pode-se observar o procedimento ordena com tamanho da entrada n-i e de complexidade nlogn e MAX, com tamanho da entrada n e complexidade n.

FIGURA 4.3 – Opção Visualizar Tabela

Para definir um novo procedimento o usuário deve acionar a opção **Inserir Item**. A janela da FIGURA 4.4 irá surgir na tela. Nela o usuário irá definir o nome do procedimento, o tamanho da entrada para este procedimento e a complexidade deste procedimento. Caso o usuário não defina a complexidade do procedimento o sistema irá assumir a complexidade do mesmo como constante (1).

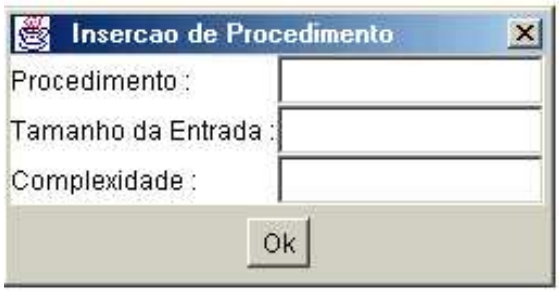


FIGURA 4.4- Opção Inserir Item

O usuário tem a possibilidade de alterar um item da tabela, para tanto deve acionar a opção **Alterar Tabela**. Quando acionada esta opção a janela representada na FIGURA 4.5 irá surgir na tela.

FIGURA 4.5 – Opção Alterar Tabela

Na opção **Alterar Tabela**, o usuário poderá modificar o nome do procedimento, o tamanho da entrada e a complexidade. Para fazer a alteração o usuário devera selecionar o procedimento a ser alterado, na opção Procedimento. Ao selecionar o procedimento o sistema irá preencher os campos Definição, Tamanho da Entrada e Complexidade com as respectivas informações. Basta que o usuário digite o novo nome, novo tamanho da entrada ou nova complexidade, e acione o botão **Alterar** para que a nova informação seja inserida na tabela. O acionamento do botão **Sair** fecha esta janela. Para excluir um item na tabela, FIGURA 4.6, o usuário deve selecionar o procedimento e acionar o botão **excluir**. A opção **Sair** irá fechar esta janela.

FIGURA 4.6 – Opção Excluir Item

Terceira Etapa:

Após o usuário ter definido pela utilização ou não da tabela e havendo um algoritmo carregado o processo de análise pode ser iniciado. Para isto, deve ser selecionada a opção **Analisar/Iniciar**, FIGURA 4.7.

FIGURA 4.7 – O Menu Analisar

Quarta Etapa:

Caso o processo de análise tenha início e não exista nenhum algoritmo carregado o sistema emite uma mensagem ao usuário informando que não existe algoritmo para ser analisado. Do contrário, o sistema inicia o processo de análise e solicita ao usuário que este informe a variável que identifica o tamanho da entrada.

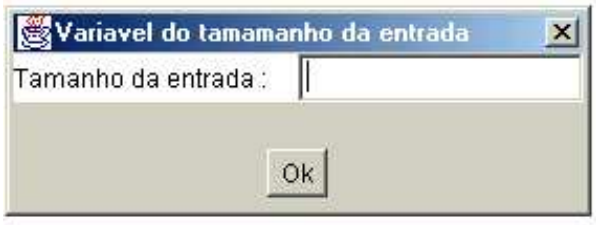


FIGURA 4.8 – Variável que Identifica o Tamanho da Entrada

Quinta Etapa:

Para poder calcular a complexidade de estruturas condicionais e iterações o sistema irá interagir com o usuário a fim de que este forneça as informações necessárias para que estas estruturas possam ser analisadas.

A Estrutura Condicional *if*

Quando o analisador léxico-sintático encontrar o comando if o sistema irá solicitar que o usuário forneça a complexidade da avaliação da sua condição. Por exemplo: if a < b then , o sistema necessita saber qual a complexidade da avaliação de a < b. Para obter esta informação o sistema apresenta ao usuário a janela da FIGURA 4.9. Caso o usuário não informe a complexidade da condição, simplesmente acione o botão **Ok** o sistema assume complexidade constante (1).

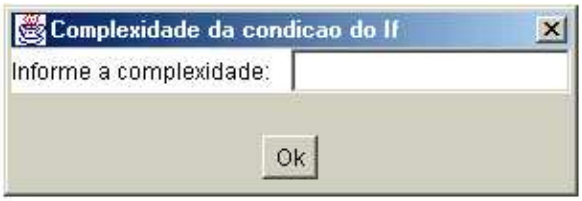


FIGURA 4.9 – Complexidade da Estrutura Condicional.

A Estrutura de Iteração Condicional *while*

Da mesma forma que atua com o comando if o sistema irá agir quando encontrar um comando while. Neste caso, porém, mais informações são necessárias. O processo de cálculo irá ocorrer na forma de iteração definida, conforme pode-se observar na seção 3.3. Desta forma, o usuário terá que informar ao sistema, um limite superior, ou seja, o valor que referente ao número de iterações que serão executadas. A FIGURA 4.10, ilustra a janela que irá solicitar essa informação.

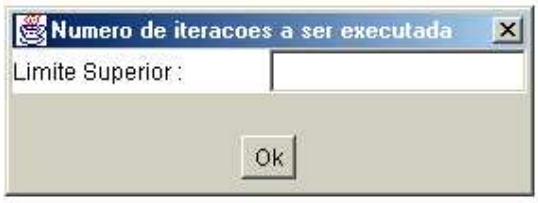


FIGURA 4.10 – Iteração Condicional

O sistema prossegue o processo de análise e irá solicitar que o usuário informe a complexidade da avaliação da condição do comando while.

De posse de todas estas informações o sistema irá concluir o processo de análise do algoritmo para o Pior Caso.

A seguir pode-se observar alguns exemplos de aplicação do sistema ANAC.

4.1 Aplicação do Protótipo ANAC

A seguir será exemplificado o uso do ANAC para a análise no pior caso de algoritmos cuja complexidade já é conhecida pela literatura referente ao assunto.

4.1.1 O Algoritmo MaxMin

Na análise deste algoritmo serão ilustrados todos os passos relativos a análise do algoritmo, incluindo os passos de interação com o usuário.

O presente algoritmo tem por objetivo determinar o máximo e o mínimo de uma tabela, armazenada como vetor **Tab**, comparando cada elemento com candidatos.

```
program MaxMin;
begin
        Max := Tab[1];
        min := Tab[1];
        for i := 1 to n-1 do
                 if Tab[i] > Max then
                         max := Tab[1];
                 endif;
                 if Tab[i] < min then
                         max := Tab[1];
                endif;
        endfor;
end.
```

Para iniciar a execução do processo de análise do algoritmo acima, o usuário deve acionar no menu **Analisar**, o item **Iniciar**. A janela da FIGURA 4.11 irá solicitar que o usuário defina a variável que identifica o tamanho da entrada.

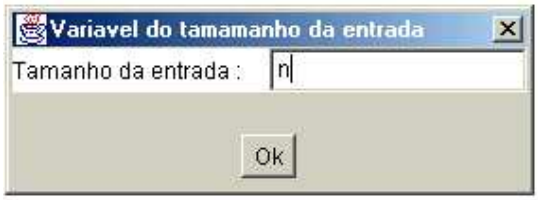


FIGURA 4.11 – Identificação do Tamanho da Entrada

Neste exemplo o usuário definiu como *n* a variável do tamanho da entrada. De posse dessa informação o sistema prossegue sua análise:

```
x1 = C ( Max := Tab[1] ) + x2 =  1  + x2
x2 = C ( min := Tab[1]  ) + x3 =  1  + x3
x3 = C ( for i := 1 to n-1 do  x4 ) + x5 = [ SUM(i = 1 , n-1) x4 ] + x5
x4 = C ( Tab[i]>Max ) + x6 + x7
```

Neste momento da análise o sistema identificou uma estrutura condicional if. Para calcular a complexidade desta estrutura o sistema solicitará ao usuário que indique a complexidade da avaliação da condição (Tab[i] > Max). Neste exemplo o usuário definiu como complexidade constante (=1). Caso o usuário não digite nenhuma informação o sistema assume a complexidade da avaliação como constante. FIGURA 4.12.

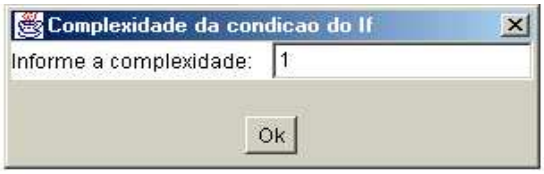


FIGURA 4.12 – Complexidade da Avaliação da Condição

x6 = C (max := Tab[1]) + x8 = 1 + x8
x8 = C (endif) = 0
x6 = 1 + 0 = 1
x7 = C (Tab[i]<min) + x9 + x10

Neste ponto novamente a janela da FIGURA 4.12 será apresentada ao usuário para que ele defina a complexidade da avaliação da condição Tab[i] < min, encontrada nesta nova estrutura condicional if.

Após esta definição o processo de análise transcorre até o final sem novas interações com o usuário, uma vez que não encontrou novas estruturas que necessitassem de informações externas para a efetivação do processo de cálculo da complexidade do algoritmo.

x9 = C (max := Tab[1]) + x11 = 1 + x11
x11 = C (endif) = 0
x9 = 1 + 0 = 1
x10 = C (endfor) = 0
x7 = 1 + 1 + 0 = 1
x4 = 1 + 1 + 1 = 1
x5 = C (end) = 0
x3 = [SUM (i = 1 , n-1) 1] + 0 = n + 0 = n
x2 = 1 + n = n
x1 = 1 + n = n
O (n)

4.1.2 O Algoritmo Classifica

Este algoritmo classifica um vetor **A** de dimensão *n*, em ordem não crescente, varrendo-o *n* vezes. Na primeira varredura, coloca o maior elemento na primeira posição mais à esquerda; na segunda varredura, coloca o segundo maior elemento na segunda posição mais à esquerda e assim por diante. Este algoritmo já está no formato Pascal-like do sistema.

```
program classi;
begin
        for i := 1 to n do
                for j := 1 to n-i do
                   if A[j] > A[j+1] then
                         Temp := A[j];
                         A[j] := A[j+1];
                         A[j+1] := Temp;
                   endif;
                endfor;
        endfor;
```

```
end.

x1 = C ( for i := 1 to n do  x2 ) + x3 = [ SUM(i = 1 , n) x2 ] + x3
x2 = C ( for j := 1 to n-i do  x4 ) + x5 = [ SUM(j = 1 , n-i) x4 ] + x5
x4 = C ( A[j]>A[j+1] ) + x6 + x7
x6 = C ( Temp := A[j] ) + x8 =  1  + x8
x8 = C ( A[j] := A[j+1] ) + x9 =  1  + x9
x9 = C ( A[j+1] := Temp ) + x10 =  1  + x10
x10 = C ( endif  )  = 0
x9 = 1 + 0 = 1
x8 = 1 + 1 = 1
x6 = 1 + 1 = 1
x7 = C (  endfor  )  = 0
x4 = 1 + 1 + 0 = 1
x5 = C (  endfor  )  = 0
x2 = [ SUM ( j = 1 , n-i ) 1 ] + 0 = n + 0 = n
x3 = C (  end  )  = 0
x1 = [ SUM ( i = 1 , n ) n  ] + 0 = n^2+0 = n^2

O ( n^2 )
```

4.1.3 O Algoritmo LOTO

Este algoritmo recebe um vetor Rslt[1..3] dando o resultado de um teste da loteria esportiva, contendo os valores 1 (coluna 1), 2 (coluna 2) e 3 (coluna do meio). Lê, a seguir uma matriz Apst[1..n,0..3], com o número do cartão de cada apostador seguido de seus 13 palpites. O objetivo é duplo: contabilizar o número de pontos feitos por cada apostador, e, para cada ganhador, escrever o número do apostador com a mensagem: "Ganhador, parábens".

```
program Loto;
begin
        i := 1;
        while i <= n do
                Ptos[i] := 1;
                for j := 1 to 13 do
                     if Apst[i,j] = Rslt[j] then
                          Ptos[i] := Ptos[i] + 1;
                     endif;
                endfor;
                i := i + 1;
        endwhile;
        for i := 1 to n do
                if Ptos[i] = 13 then
                        write ('Apst[i,0],Ganhador, parabens'  );
                endif;
        endfor;
end.

x1 = C ( i := 1 ) + x2 =  1  + x2
```

x2 = C (while i <= n do x3) + x4 = C (i<=n) + [n.(C (i<=n) +x3)] + x4
x3 = C (Ptos[i] := 1) + x5 = 1 + x5
x5 = C (for j := 1 to 13 do x6) + x7 = [SUM(j = 1 , 13) x6] + x7
x6 = C (Apst[i,j]=Rslt[j]) + x8 + x9
x8 = C (Ptos[i] := Ptos[i] + 1) + x10 = 1 + x10
x10 = C (endif) = 0
x8 = 1 + 0 = 1
x9 = C (endfor) = 0
x6 = 1 + 1 + 0 = 1
x7 = C (i := i + 1) + x11 = 1 + x11
x11 = C (endwhile) = 0
x7 = 1 + 0 = 1
x5 = [SUM (j = 1 , 13) 1] + 1 = 13 + 1 = 1
x3 = 1 + 1 = 1
x4 = C (for i := 1 to n do x12) + x13 = [SUM(i = 1 , n) x12] + x13
x12 = C (Ptos[i]=13) + x14 + x15
x14 = C (write (Apst[i,0],Ganhador, parabens) + x16 = 1 + x16
x16 = C (endif) = 0
x14 = 1 + 0 = 1
x15 = C (endfor) = 0
x12 = 1 + 1 + 0 = 1
x13 = C (end) = 0
x4 = [SUM (i = 1 , n) 1] + 0 = n + 0 = n
x2 = 1 + [n.(1 + 1)] + n = n
x1 = 1 + n = n

O (n)

5 Análise Automática de Algoritmos

Este capítulo irá abordar alguns dos principais estudos feitos na área de análise automática de algoritmos, apresentando os sistemas Metric, ACE, Lambda-Upsilon-Omega e ACME. Serão apresentadas as características destes sistemas, suas aplicações e um comparativo destes sistemas com o sistema ANAC proposto neste trabalho.

5.1 O Sistema Metric

Em 1975, Ben Wegbreit desenvolveu o protótipo de um sistema cujo objetivo era tornar o processo de análise de algoritmos mecânico. O sistema foi denominado METRIC [WEG 75].

Wegbreit, encontrou, como uma importante limitação ao desenvolvimento do sistema, o fato da análise de algoritmos requerer considerável conhecimento matemático. Segundo Weigbreit, um sistema especialista para análise automática de algoritmos deveria necessariamente incluir todas as técnicas apresentadas no trabalho de Knuth [KNU 68].

Apesar da limitação, Wegbreit estabeleceu a análise mecânica de programas como uma atividade viável. Este fato constituiu-se em uma inovação nesta área de pesquisa e motivou o desenvolvimento de outros trabalhos de pesquisa com esta finalidade.

A análise mecânica de programas possui três aplicações principais:

1. Como uma ferramenta de engenharia de software: auxiliando programadores na compreensão do comportamento de um programa.
2. Em sintetizadores automáticos de programas.
3. Na compilação de sistemas para linguagens de alto nível.

O METRIC é um protótipo de sistema que foi construído para mecanizar a análise de programas, como tal, ele se concentra nas características principais do algoritmo, ignorando certos detalhes periféricos. As medidas de desempenho analisadas pelo METRIC são as medidas apresentadas em [KNU 68], que são a complexidade no melhor caso, pior caso e caso médio (<*min* ,*max*, *mean*, *variance*>). Estas medidas são obtidas segundo uma microanálise dos programas [COH 82]. Isto torna o sistema METRIC mais preciso do que os sistemas que realizam macroanálise de programas, uma vez que estes estão voltados a análise assintótica das medidas. Entretanto, a microanálise de programas utilizada pelo METRIC pode gerar fórmulas intratáveis e de pouca utilidade prática para o analista.

O sistema METRIC realiza análise de programas escritos na linguagem Lisp, basicamente programas para manipulação de listas. Maiores detalhes sobre programas para processamentos de listas em Lisp podem ser encontrados em [McC 60] e [WEI 67]. O sistema foi desenvolvido na linguagem Interlisp.

5.1.1 Estrutura Interna do Sistema

O METRIC possui as estruturas organizadas, conforme pode ser observado na FIGURA 5.1. Linhas sólidas mostram o fluxo de controle através do sistema; linhas pontilhadas mostram a utilização dos dados, dados de entrada ou resultados provisoriamente computados. As bases de entradas de dados são um conjunto de definições de procedimentos e um conjunto de tabelas usados para estabelecer os custos simbólicos das operações elementares.

A entrada do sistema é um programa e a medida na qual o programa será analisado. A saída é a análise especificada e uma base de dados do desempenho dos procedimentos analisados neste processo.

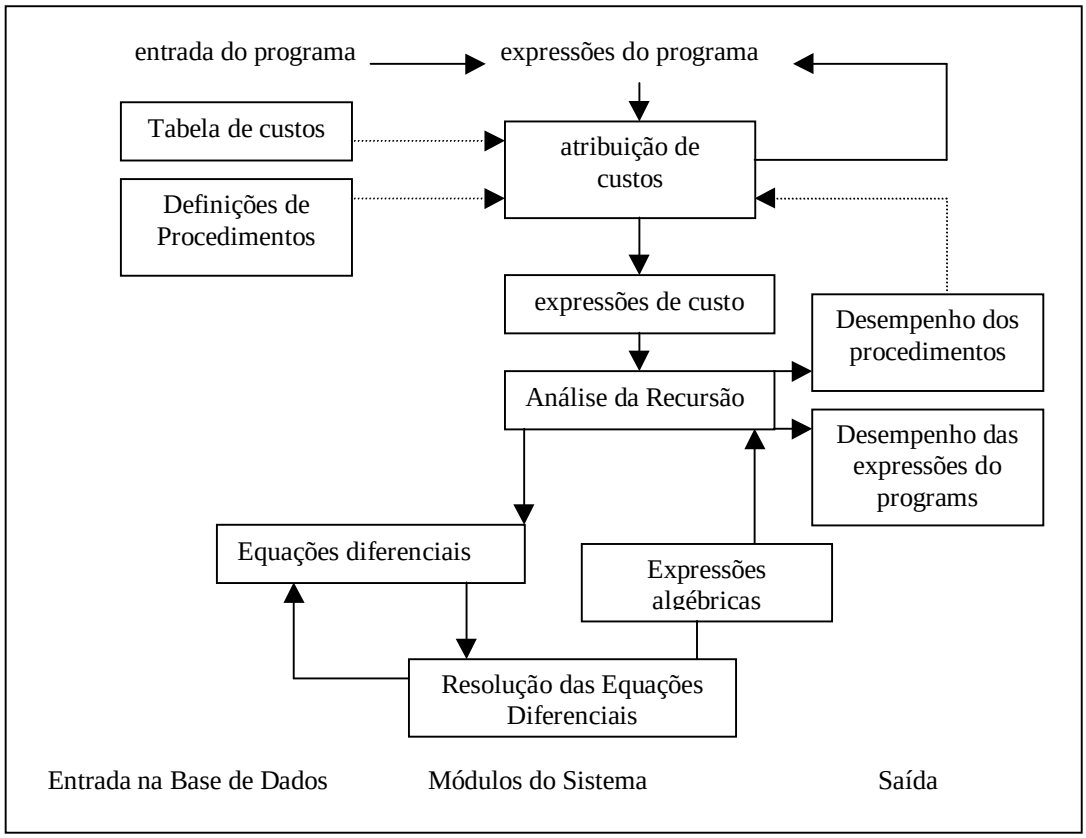


FIGURA 5.1 – A estrutura interna do METRIC

A análise de um sistema se dá em três fases:

Fase 1: Atribuição de custos locais. Um *custo* é atribuído às partes do programa da seguinte forma: Para operações primitivas (CAR, por exemplo) e chamadas de funções, são atribuídos custos definidos pela tabela de custos. Para as demais operações definidas, são atribuídos os custos obtidos na sua definição, exceto para chamadas recursivas de procedimentos que são detectadas e tratadas de maneira diferente. A análise de um procedimento não recursivo é determinado pela composição de custos locais. Esta fase mapeia um programa para uma expressão simbólica de custos, que especifica seu custo para uma dada medida. Funções recursivas são passadas para a próxima fase.

Fase 2: Análise de recursão. O procedimento é simbolicamente avaliado para determinar como as variáveis recursivas mudam de uma chamada para outra. Isto fornece a estrutura recursiva de seqüência de computação. A seguir, a seqüência de computação é projetada nos inteiros pela construção de um conjunto de equações diferenciais.

Fase 3: Solução das equações diferenciais. Uma ou mais, das seguintes técnicas, são usadas para avaliar as expressões: eliminação de variáveis, análise no pior/melhor caso, derivação de funções geradoras, dentre outras.

A solução das equações diferenciais fornece uma expressão para o desempenho do procedimento recursivo original. Esta solução é simplificada, colocada na forma funcional, e armazenada sob o par <*procedure*, *measure*> para que possa ser recuperada posteriormente.

Na apresentação do programa exemplo, será utilizada a seguinte notação:
{}- representa uma lista vazia.
{*f*.*r*}- representa uma lista não vazia, onde *f* é a cabeça da lista e *r* é o rabo da lista.
CONS(*f*,*r*) = {*f*.*r*}
CAR({*f*.*r*}) = *f*
CDR({*f*.*r*}) = *r*
ATOM(*x*) é um predicado que é verdadeiro se e somente se *x* não é uma lista ou a lista está vazia.
NULL(*x*) é definido como *x* = {}.
Expressões condicionais são escritas como:
if p_1 **then** e_1 **else if** p_2 **then** e_2...**else** e_n

Dada a definição de um conjunto de procedimentos, o METRIC tenta gerar a análise para: o tempo de execução do algoritmo, número de vezes que a operação *CONS* é executada, número total de células presentes na lista após a obtenção do resultado, dentre outros.

O sistema METRIC é capaz de tratar apenas programas simples, como os que podem ser utilizados como exercício introdutório da programação Lisp (em geral, *append*, *reverse*, *nth*, *substitute*, *flatten*, *member*, e *union*). Programas gerados para obtenção de um elemento da lista, substituição de um elemento da lista, inversão dos valores da lista, união de listas, dentre outros.

5.1.2 Exemplo de Aplicação do Sistema

A seguir são ilustrados exemplos de aplicação do METRIC, apresentados em [WEG 75].

Exemplo 1:

O procedimento *FLAT* é utilizado para compactar listas, construindo uma lista de nível atômico partindo de uma lista possivelmente multi-nível, em geral, *FLAT*({*A*.{{*B*.*C*}.*D*}}) = {*A*,*B*,*C*,*D*}. Uma maneira de fazer isto é usar uma função auxiliar *FLAT*2:

FLAT(*L*) ≡ *FLAT*2(*L*,{})

*FLAT*2(*X*,*Y*) =
if *ATOM*(*X*) **then** *CONS*(*X*,*Y*)
else *FLAT*2(*CAR*(*X*),*FLAT*2(*CDR*(*X*),*Y*)

O METRIC determina que o tempo de *FLAT*(*L*) dependa do *size* (tamanho) de *L*, ou seja, o número de células da lista. Especificamente, o tempo é dado por $c_0 + c_1.s$, onde *s* é o *size* de *L*, e:

c_1 = *cons* + 2.*fncall* + 2.*cr* + 2.*atom* + 7.*vref*.

$c_0 = cons + atom + fncall + 4.vref + cref$.

Na análise de um programa sob determinada medida, uma ou mais medidas são tipicamente aplicadas na decomposição. Por exemplo, na computação do *tempo* de *REVERSE*(*FLAT*(*APPEND*(*P*,*Q*))), o *length* (comprimento) do argumento de *REVERSE* é necessário. Analisando *FLAT* sob a medida *length*, o METRIC obtém *length*(*FLAT*(*L*)=1 + *size*(*L*). A seguir, obtém-se *size*(*APPEND*(*P*,*Q*)) = *size*(*Q*). Consequentemente, o tamanho do argumento para *REVERSE* é dado por 1 + *size*(*P*) + size(*Q*).

Length (comprimento) e *size* (tamanho) são propriedades estruturais de variáveis, semelhantes a dimensões de vetores ou número de registros em um arquivo. O METRIC tenta expressar o comportamento de programas nestes termos. Quando isto não é possível, devido a testes internos não relacionados às propriedades estruturais, o METRIC expressa a análise com desempenho no qual probabilidades de testes não avaliáveis aparecem como parâmetros.

Exemplo 2:

O número de vezes que o átomo X aparece no topo da lista *L* é computado por:

COUNT(*X*,*L*) ≡
if (*NULL*) **then** 0
else if *X* = *CAR*(*L*)
then *ADD*1(*COUNT*(*X*,*CDR*(*L*)))
else *COUNT*(*X*,*CDR*(*L*))

A probabilidade do teste *X* = *CAR*(*L*) ocorrer é necessária para se obter o tempo de desempenho de *COUNT*. Entretanto, a operação "=" é primitiva e não pode ser analisada. O sistema não pode continuar sem utilizar informações adicionais fornecidas a ele. Se o sistema acessar sua base de dados irá verificar que o teste "=" em *COUNT* pode ser tratado como constante. O sistema atribui então uma propriedade simbólica, denominada *p* cujo valor será determinado pela medida. O METRIC determina então que o tempo de desempenho é dado por:

$$< c_0 + c_1 n, c_0 + c_2 n, c_0 + c_3 n, c_4 n >$$

onde, $n = length(L)$ e

$c_0 = null + cref + vref$
$c_1 = fncall + null + eq + 2.cr + 5.vref$
$c_2 = add1 + 2.fncall + null + eq + 2.cr + 5.vref$
$c_3 = p.add1 + p.fncall + fncall + null + eq + 2.cr + 5.vref$
$c_4 = p.add1^2 + 2.p.add1.fncall + p.fncall^2 - p^2.add1^2 - 2.p^2.add1.fncall - p^2.fncall^2$

Exemplo 3:

O procedimento *UNION* assume como argumentos listas de símbolos atômicos não repetidos e formam o conjunto união:

UNION(*X*,*Y*) ≡
if *NULL*(*X*) **then** *Y*
else if *MEMBER*(*CAR*(*X*),*Y*)
then *UNION*(*CDR*(*X*),*Y*)

else *CONS*(*CAR*(*X*), *UNION*(*CDR*(*X*),*Y*))

Aqui o teste a ser tratado probabilisticamente é um procedimento definido:

MEMBER(*Z*,*L*) ≡
if *NULL*(*L*) **then false**
else if *Z* = *CAR*(*L*) **then true**
else *MEMBER*(*Z*,*CDR*(*L*))

de maneira que *probability*(*MEMBER*(*CAR*(*X*),*Y*)) é uma expressão derivada que pode ser expressa em termos de outras quantidades. Analisando *MEMBER* usando o resultado da expressão *probability* como sendo **true**, o METRIC obtém 1 - (1- *a*)^{m} onde *a* = *probability* (*Z* = *CAR*(*L*)) e *m* = *length*(*L*). Usando isto, o *size* de *UNION*(*X*,*Y*) é dado por:

$$<m,\ m + n,\ m + n.(1 - a)^{m},\ n.(1 - a)^{m} - n.(1 - a)^{2.m}>$$

onde, *m* = *lenght*(*Y*) e *n*= *lenght*(*X*).

5.2 O Sistema ACE

O sistema ACE (*Automatic Complexity Evaluator*) [MET 88] é capaz de analisar programas tais como: programas de ordenação, programas numéricos, etc, de forma totalmente mecânica. Partindo-se do programa inicial, uma função de complexidade de tempo é derivada. Esta função é transformada em uma função não recursiva equivalente, de acordo com o princípio de indução de recursão de McCarthy [McC 63], usando uma biblioteca pré-definida de definições recursivas.

O comportamento de um programa pode ser caracterizado de vários modos. As abordagens são geralmente classificadas em duas categorias: microanálise e macroanálise. A microanálise tem por base o tempo necessário para executar cada operação elementar envolvida no programa, ou seja, é uma análise empírica. A macroanálise é uma medida do comportamento assintótico do programa. A microanálise provê uma medida mais precisa de complexida, mas pode produzir fórmulas intratáveis. Por estar voltado ao comportamento assintótico, a abordagem presente no ACE é a macroanálise.

O sistema ACE utiliza uma versão de programas funcionais [BAC 78] como sua linguagem fonte. Ao contratário do METRIC, que gera equações diferenciais a partir do programa original, o ACE apenas manipula programas funcionais. O princípio de funcionamento do ACE é o de permanecer no campo das linguagens funcionais para poder se utilizar da álgebra de programas associada as linguagens funcionais. O método consiste na derivação da função recursiva inicial *f* em uma função recursiva *Cf* cujo valor é a complexidade de *f*. Partindo desta equação recursiva, o sistema tenta transformar esta equação recursiva em uma equação não recursiva, utilizando o formalismo das linguagens funcionais e alguns modelos pré-definidos. A característica principal do sistema está na transformação das equações recursivas em equações não recursivas, que é o que irá definir qual a complexidade do programa. Para esta tarefa é utilizado o princípio de indução de recursão de McCarthy [McC 63]. Quando o sistema consegue fazer, com sucesso, a conversão das equações a expressão final de complexidade será uma composição de funções bem conhecidas, tais como funções exponenciais, logarítmicas, etc. O sistema ACE, segundo Le Metayer [MET 88], é

capaz de tratar um grande número de programas (programas numéricos, programas de ordenação, etc) e o objetivo é analisar uma grande escala de aplicações.

5.2.1 Visão Geral do Sistema

Para Métayer, os resultados que esperava-se obter do sistema seria algo como: "A complexidade da função fatorial é uma função linear sobre o argumento", ou "A Complexidade da função *selection-set* é o quadrado do tamanho do argumento." No entanto, para poder fornecer o resultado desta maneira dois problemas principais podem ser observados:

Determinar a propriedade relevante do argumento, ou seja, definir a propriedade estrutural da qual a complexidade depende. Por exemplo, a complexidade de um problema de ordenação depende do tamanho do argumento, ao passo que a complexidade do fatorial depende do valor deste argumento. Esta relevante propriedade é denominada "medida."

Análise de recursão. Deve-se obter uma expressão não-recursiva da função de complexidade em termos da medida. Esta função de complexidade deve ser uma composição de funções numéricas bem conhecidas tais como logaritmos, exponenciais, identidade, etc.

É relativamente simples obter-se uma definição recursiva da função de complexidade, o principal problema é transformá-la em uma definição não-recursiva. Métayer usou em seu sistema, como tentativa de solucionar este problema, uma série de axiomas da álgebra de programas dos programas funcionais [BAC 81]e [WIL 82].

O objetivo do ACE é a macroanálise de programas, por isto não é considerado o custo de funções primitivas e formas funcionais; desta forma a complexidade assintótica e o número de chamadas recursivas necessárias para a avaliação do programa possuem a mesma ordem de magnitude.

O método consiste na derivação da função recursiva inicial *f* em uma função recursiva de complexidade *Cf*. *Cf* assume o mesmo argumento de *f*, mas o resultado é um valor inteiro que representa o número de chamadas recursiva necessárias para a avaliação da *f* aplicada a este argumento. O primeiro passo do sistema é derivar *Cf* de *f*. Esta transformação é obtida através da tradução dirigida por sintaxe do programa original. Alguns axiomas dos programas funcionais são utilizados para realizar simplificações e transformações do programa; o passo final é a aplicação do princípio de indução de recursão para se obter uma função não recursiva. O princípio de indução da recursão pode ser definido como: "duas funções que verificam a mesma equação recursiva são equivalentes sobre o domínio da função definida para esta equação."

Como já foi citado anteriormente, a principal etapa do processo de análise do sistema ACE é a transformação das expressões recursivas em não recursivas. É esta etapa que irá fornecer a complexidade do programa analisado. Para que esta etapa possa ser realizadas cinco operações são aplicadas:

Simplificação: Para poder aplicar a simplificação o sistema possui em sua base de dados uma biblioteca de axiomas de programas funcionais.

Fatorização: Quando não é possível aplicar o princípio de indução de recursão, o ACE tenta colocar a função de complexidade *Cf* na forma C *o m*, onde *m* a medida da função.

Particionamento: A operação de particionamento é útil quando o programa contém testes que não são relevantes para a avaliação da complexidade. Quando a operação de fatorização falha o sistema irá particionar a equação em *n* equações. O sistema é então chamado para resolver, recursivamente, estas equações. O objetivo nesta operação é tornar o processo de resolução mais simples.

Aplicação do princípio de recursão: O objetivo das operações anteriores é gerar uma equação recursiva de acordo com algum modelo pré-definido existente na base de dados do sistema. Para poder realizar esta geração o sistema é suprido com uma biblioteca de definições recursivas.

Substituição: Quando não é possível aplicar o princípio de recursão o ACE tenta transformar a expressão em uma expressão aproximada, que esteja presente em sua biblioteca, desde que esta expressão preserve a ordem de magnitude das expressões envolvidas. Como o sistema avalia a complexidade assintótica, no pior caso, esta substituição é possível. Caso não seja possível fazer a substituição das expressões a análise do sistema falha.

Para a transformação das equações nem sempre é necessário a utilização de todas as operações anteriormente citadas. Como pode-se observar na FIGURA 4.2, na maioria das vezes uma operação só será executada se na operação precedente tiver havido falha, neste caso, mais refinamentos são necessários, e desta forma, passa-se a expressão para a operação subsequente para tentar obter a expressão final.

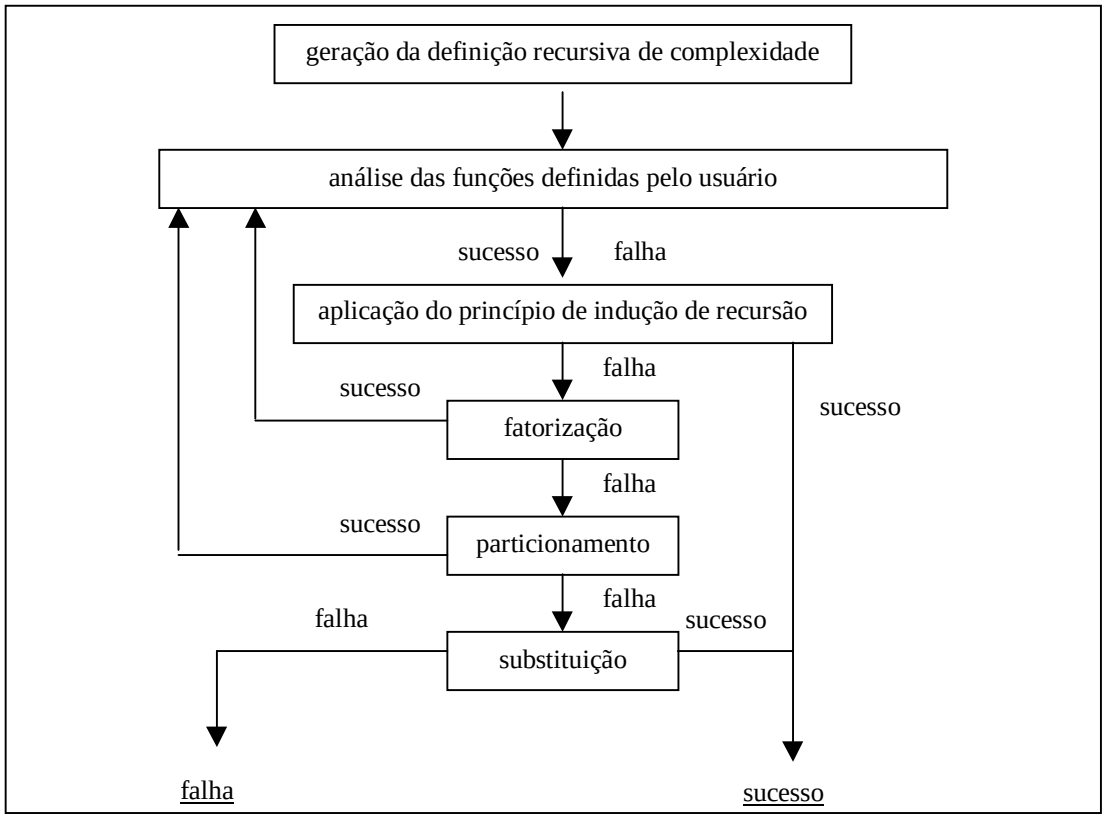


FIGURA 5.2 – Organização do sistema ACE

5.2.2 Exemplo de Aplicação do sistema ACE

Esse exemplo foi apresentado no artigo [MET 88] e é reproduzido nesta seção. Neste exemplo apenas algumas etapas do processo de análise são ilustrados. Considere *f* uma função tal que:

$$f = \text{eq0} \rightarrow \text{"0"}; \text{plus1 } o \text{ } f \text{ } o \text{ sub1}.$$

onde:

eq0 (x) := x = 0;
plus1 (x) := x + 1;
sub1 (x) := x - 1;
o é composição.

Considerando-se que a função identidade também apresenta estas características, obtém-se:

$$n \geq 0 \Rightarrow f{:}n = \text{id}{:}n = n.$$

Para exemplificar a utilização do sistema deseja-se avaliar a complexidade de uma função fatorial, definida por:

$$\text{fact} = \text{eq0} \rightarrow \text{"1"}; *o[\text{id, fact } o \text{ sub1}].$$

Isto é:

fact(n) = eq0(n) → "1",
*o[id, fact *o* sub1](n) =
= fact(n) = n = 0 → 1, n*fact(n-1).

Assume-se que *eq0* e *sub1* são funções primitivas. É dado apenas o resultado das duas principais fases do sistema. O primeiro passo produz a seguinte equação:

*C*fact = eq0 → +*o*["0", "0"];
+*o*["0",+*o*["0",+*o*["0", +*o*[plus1 *o* *C*fact *o* sub1, "0"]]]].

onde:

+*o*["0", "0"], indica a composição da operação de adição com os símbolos "0".
+*o*[plus1 *o* *C*fact *o* sub1, "0"] essa representação irá compor:
plus(*C*fact(sub(n)) + "0") que a seguir será composta com os símbolos "0" anteriores.

As complexidades das funções primitivas são definidas como "0"; *C*fact *o sub1* é o número de chamadas recursivas introduzidas por *fact o sub1*, e *plus1* vale para a chamada inicial. Após a execução da operação de simplificação realizada com base no seguinte axioma:

+*o*["0", *f*] = *f* e +*o*[*f*,"0"] = *f*, o sistema obtém:

*C*fact = eq0 → "0"; plus1 *o* *C*fact *o* sub1,
*C*fact(n) = n = 0 → 0, 1 + *C*fact(n-1)

e o principio de indução de recursão faz com que o sistema conclua que:

*C*fact = id, i.é.,
*C*fact(n) = n

sobre o domínio dos inteiros positivos, ou seja, a complexidade da função fatorial é linear sobre o argumento.

Este exemplo baseou-se, principalmente, na abordagem da transformação para obter o resultado da análise.

5.3 O Sistema Lambda-Upsilon-Omega

O Lambda-Upsilon-Omega, ΛϒΩ [FLA 88], é um sistema desenvolvido para analisar, de forma automática, classes bem definidas de algoritmos e estruturas de dados. A medida de análise empregada pelo sistema é a análise no caso médio.

O sistema ΛϒΩ baseia-se na conjunção de duas principais idéias:

Pesquisando-se metodologias de análise combinatorial, encontrou-se uma relação entre, estruturas de dados e especificações de algoritmos, e equações que operam sobre funções geradoras. Esta relação foi aplicada no sistema. Verificou-se que os programas analisados recaiam em uma categoria bem definida de programas que podiam ser decompostos em estruturas de dados, que por sua vez também podiam ser decompostas. Foram desenvolvidos, então, e inseridos no sistema Lambda-Upsilon-Omega dois sub-sistemas: o *Sistema Analisador Algébrico – ALAS* e o *Sistema Analisador Analítico – ANANAS.* A tarefa do sistema ALAS é a de decompor os algoritmos em estruturas de dados e, a seguir, decompor as estruturas de dados, gerando desta forma, equações, que serão passadas para o sistema ANANAS para o cálculo final da complexidade do algoritmo. O sistema ALAS foi implementado na linguagem CAML, que é uma extensão da linguagem ML e possui cerca de 2.000 linhas de código.

As equações obtidas pelo sistema ALAS são passadas para o sistema ANANAS, que constitui-se em uma coleção de rotinas algébricas escritas na linguagem Maple [CHA 88], compreendendo cerca de 5.000 linhas de código. O propósito deste sub-sistema é extrair a forma assintótica dos coeficientes das funções geradoras. O analisador analítico (assintótico) implementa uma coleção de poderosas estratégias baseadas na análise de complexidade de algoritmos.

Em essência, qualquer programa especificado no sistema Lambda-Upsilon-Omega pode ser analisado automaticamente em termos de funções geradoras, desde que as regras estejam completas em relação à linguagem.

O sistema implementa como tipos básicos de dados, árvores de termos, como encontradas nos sistemas de álgebra simbólica. O analisador pode tratar grandes classes de funções com expressões explícitas. Deste modo, o sistema Lambda-Upsilon-Omega pode gerar, cerca de uma dúzia de análises não-triviais de caso médio de algoritmos como: diferenciação formal, algumas simplificações algébricas e algoritmos de *matching*. O analisador analítico pode determinar expansões assintóticas para grandes classes de funções geradoras que irão resultar na análise do algoritmo.

A implementação do Lambda-Upsilon-Omega foi iniciada em meados de 1987 segundo seus desenvolvedores [FLA 88] não tem nenhuma pretensão de ser universal.

Um algoritmo para ser analisado pelo Lambda-Upsilon-Omega deve ser descrito na linguagem da Linguagem de Descrição de Algoritmos (*Algorithm Description Language* - Adl). A Adl é uma linguagem cujas primitivas correspondem aos mecanismo de tipos de dados estruturados.

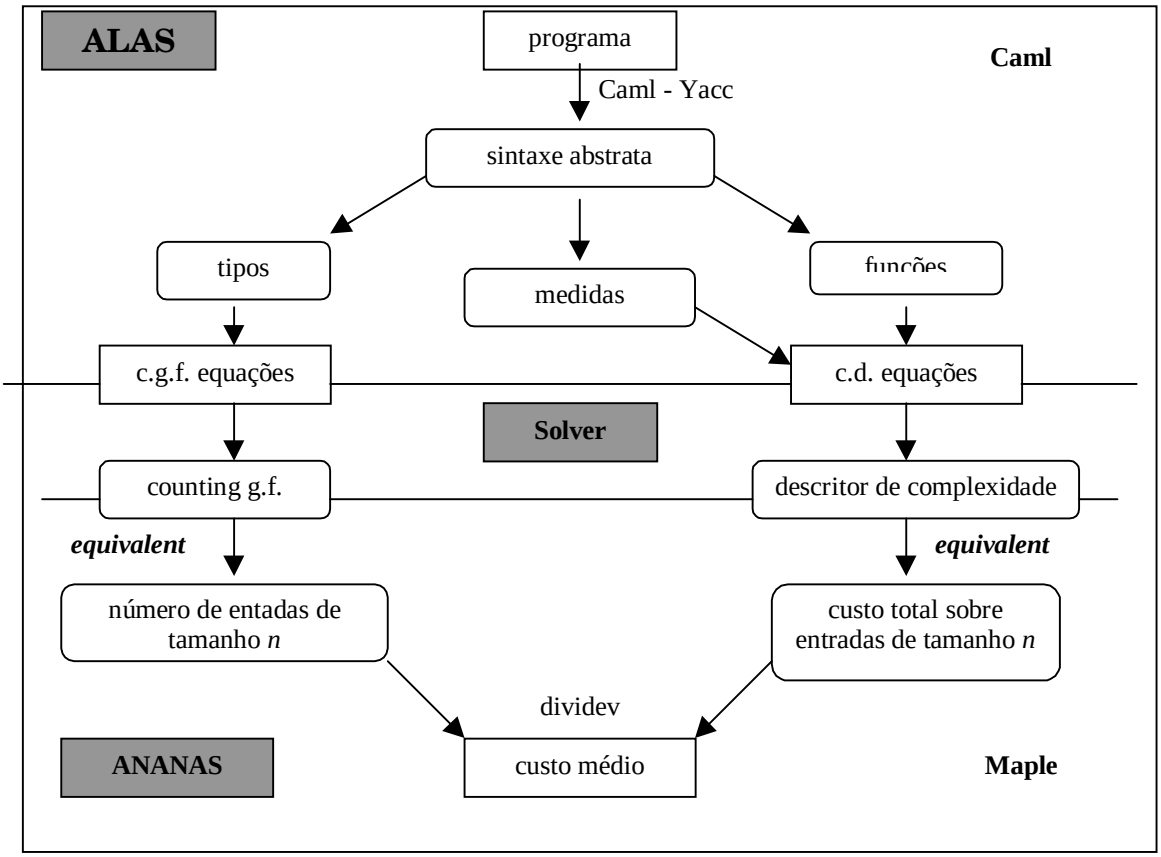


FIGURA 5.3 - A estrutura interna do sistema Lambda-Upsilon-Omega

5.3.1 Uma Sessão de Exemplo

A FIGURA 5.4 mostra a inicialização, utilização e análise efetuada pelo sistema Lambda-Upsilon-Omega, retirada de [FLA 88].

```
Script started on Fri Jan 29 12:08:24 1988
% maplelisp [Inicicialize Lisp and Maple]
; Le-Lisp (by INRIA) version 15.21 (25/Dec/87) [sun]
      |\  ^  /|         INRIA – Rocquencout
._|\ |        | /|_.   Version 4.1 --- May 1987
  \  MAPLE  /     For on-line help, type help( );
 <------ ------>
         |
[Load Algebric Analyser, Alas]
? (load luo)
Function to analyse : (diff copy1 distrib size simpl count1 filter copy evalf diffcp)
? diff [Diff with subexpression sharing]
(de diff (x)
     (caseroot * ('+ (maketree '+
     [etc...]
Analysis of diff on trees formed with ...
     symbol x of arity 0 etc...
Calling analyse...
? (maple)
*** Initial system *** [Equations Produced by Alas]
phi = 2 + 3 z 2 + z, t = 2 + 3 z t 2 + z t,
```

```
   tau_diff = 21 z t^2 + 6 z t tau_diff + 4 z t  + tau_diff  + 2 z
*** Explicit expression(s) ***  [etc...]
[Start Analytic Analyser,ANANAS]

> tn := equivalent(t); [Asymptotic number of inputs: 16 sec on sun 3/60]
                                1/4              1/2  1/2                              1/2  (-n)
                               42      (13 – 2 42    )                      (13 – 2 42     )
tn := - 1/6 ----------------------------------------------------------+ O (-----------------------)
                                1/2      3/2    1/2              1/2                  5/2
                     (-13 + 2 42   )   n       Pi       (13 – 2 42    )              n
```

```
> [etc...] [average diff time]
                                                    1/2
                                          n (- 240 + 37 42     )
                      av_diffn := 8 ---------------------------------- + O(1)
                                                  1/2       1/2
                                          (-13 + 2  42      ) 42
> evalf(av_diffn) ; [Floating point eval]
                                            6.82942 n + O(1)
> suspend_ maple;
? (analyse 'diffcp) [Diff with copy of arguments]
> [etc...]
> evalf(av_diffcpn);
                                                  3/2
                                          16.6226 n     + O (n)
> suspend_maple;
? (end) [The whole thing!]
Que Le-Lisp soit avec vous.
% ^D
script done on Fri Jan 29 12:44:51 1988
```

FIGURA 5.4 – Sessão de Exemplo do Lambda-Upsilon-Omega

Para analisar um programa no sistema Lambda-Upsilon-Omega o primeiro passo é iniciar uma sessão do sistema. Para isto é necessário chamar um *script* (usualmente no ambiente Unix) que irá iniciar uma sessão Lisp e Maple. Que é o que pode ser observado nas primeiras linhas da FIGURA 5.4, que informa que o *Script* foi iniciado, a data e hora da execução, informa que o *joint* Maple/Lisp foi carregado e fornece mais algumas informações de *status*, como: local, data e hora.

A seguir deve se carregar o subsistema ALAS e aplicá-lo ao programa a ser analisado. Essa operação é realizada com o comando:

? (load Luo)

Neste momento o sistema estará apto a receber o algoritmo a ser analisado.

O exemplo considerado é o de um programa que computa derivadas simbólicas (sem simplificação) de expressões (termos, árvores). Este processo está representado na FIGURA 5.4 como:

```
Function to analyse : (diff copy1 distrib size simpl count1 filter copy evalf diffcp)
? diff [Diff with subexpression sharing]
```

```
(de diff (x)
    (caseroot * (‘+ (maketree ‘+
    [etc...]
```

O campo [etc...] é um comentário inserido por [FLA 88] e indica que o algoritmo não foi totalmente representado, e sim, apenas parte dele para uma exemplificação simples do sistema no processo de análise.

Os passos realizados na análise do sistema são:

1º) A definição recursiva do tipo “termo” é refletida por uma equação quadrática para sua função geradora $t(z)$.

2º) A estrutura recursiva da procedure Diff[1] é refletida por uma equação linear para o descritor de complexidade $\Upsilon diff$ (função geradora de custo médio).

Estes dois passos são realizados automaticamente pelo ALAS, que também usa uma pequena rotina Maple para derivar expressões explícitas. Este processo irá gerar uma série de equações, que serão passadas para o Maple inicializado com o subsistema ANANAS. Tal inicialização se dá através do comando:

? (maple)

O ALAS gerou as expressões phi, t e tau_diff. Estas são as expressões que serão passadas para o sistema ANANAS, que será usado para operar sobre as expressões obtidas.

3º) Expansões singulares locais são então determinados.

4º) Usando teoremas gerais da análise assintótica de complexidade, expansões singulares podem ser transformadas, automaticamente, em expansões assintóticas de coeficientes. Isto é realizado por meio do comando “**equivalent”** do ANANAS. Esse processo ocorre através do comando:

> tn := equivalent(t);

Dividindo a forma assintótica dos coeficientes de $[z^n]$ em $t(z)$ e $\Upsilon diff(z)$, obtém-se a complexidade média assintótica da diferenciação simbólica, ou no formato algébrico, ou na forma de ponto-flutuante. Na FIGUARA 3.4 existe as duas representações. Após executar o comando “**equivalent”** o sistema ANANAS gera a expressão no formato algébrico. O comando *evalf* do Maple é usado para simplificar a expressão e fornecer o resultado no formato de ponto-flutuante, que são as finais do exemplo.

O comando:

?(end)

irá encerrar a execução do Maple e pressionado-se as teclas <Ctrl> + D, o *script* será encerrado.

5.4 O Sistema ACME

O ACME - Analisador de Complexidade Média [SIL 98] é um protótipo de uma ferramenta para o cálculo da complexidade de algoritmos que baseia-se na

[1] Diff é uma procedure Maple para a realização de cálculos diferencias.

metodologia de cálculo de complexidade média proposta em [ROS 97] que tornou o processo de cálculo da complexidade média mecânico e aplicável a qualquer algoritmo de forma totalmente genérica e sistemática.

O sistema ACME é basicamente um programa de análise-síntese de programas fontes escritos em uma linguagem pré-estabelecida. Este sistema tem como componentes analisadores léxico e sintático, analogamente a um compilador, mas sua síntese não é uma listagem de código ou conjunto de operações, mas sim o desenvolvimento de um cálculo baseado nas estruturas presentes no programa fonte, gerando como resultado uma a equação de complexidade.

5.4.1 Características de Funcionamento do ACME

O ACME analisa as estruturas em profundidade, nomeando as equações envolvidas no cálculo de acordo com o aninhamento, de maneira que seja necessária somente uma passada no arquivo de descrição.

Quando o analisador encontra uma estrutura como o *while*, por exemplo, ele realiza primeiramente todo o cálculo envolvido nas estruturas aninhadas dentro deste *while*, desmembrando todos os componentes que serão envolvidos no cálculo da sua equação. Desta maneira, ele vai mostrando o cálculo e realizando o encaixe de todas as equações em uma só equação, que será então a complexidade final do algoritmo. O algoritmo pr1 exemplifica este processo de análise. Neste algoritmo a iteração condicional *while* contém uma chamada ao procedimento *rotina1* e a iteração não condicional *for* contém duas chamadas a procedimentos, sendo uma chamada ao procedimento *rotina2* e uma chamada ao procedimento *rotina3*.

```
program pr1;
begin
   while ident=true do
      rotina1;
   for id:=45 to 50 do
      begin
         rotina2(id);
         rotina3(id+constante);
      end;
end.
```

O ACME sempre faz a busca em profundidade procurando a primeira estrutura mais aninhada, que neste exemplo específico é a chamada *rotina1*. Esta chamada está num terceiro nível de aninhamento e como foi a primeira neste nível a ser identificada, tem sua complexidade identificada como X3'1. Após esta identificação, o sistema identifica a estrutura que contém a chamada a este procedimento, que neste caso é um comando *while*. Este comando está no segundo nível de aninhamento, e foi a primeira estrutura a ser identificada neste nível, tendo portanto sua complexidade identificada como X2'1. Logo após são encontrados *rotina2(id)*, que é identificada como X4'1 e

rotina3(id+constante), que é identificada como X4’2 e, então, a seqüência que as contém, identificada como X3’2, a seguir é encontrado o comando *for* que contém esta seqüência, identificado como X2’2 e finalmente a seqüência que contém todo o programa que é identificada como X1. Como é possível notar, o processo ocorre através de uma estratégia de análise léxico-sintática *buttom-up*.

O desenvolvimento da análise vai sendo escrito em um arquivo, que só terá conteúdo válido caso a análise seja bem sucedida. As mensagens de erro ou sucesso vão sendo escritas em um outro arquivo.

5.4.2 A Linguagem de Descrição dos Algoritmos

O algoritmo a ser analisado pelo ACME deve ser escrito obedecendo a uma gramática Pascal-like estabelecida.

Um programa pode conter chamadas a pseudo-subrotinas e pseudo-funções, que podem ter valores (ou pesos) associados a estes, sendo que estes valores devem ser especificados pelo usuário durante a especificação do algoritmo. Por exemplo, na FIGURA 5.5, na chamada da sub-rotina *empilha(pilharecursao,1,n)#5*; O valor “5”, precedido do caracter “#”, indica que a chamada a sub-rotina *empilha* possui custo “5”. Quando o usuário desejar definir o custo de uma operação deve colocar o caracter “#” e ao lado o valor do custo.

Cada procedimento e função deve ser analisado de forma separada, sempre obedecendo as regras da gramática estabelecida. O foco da análise esta voltado para as estruturas algorítmicas que são cobertas pela metodologia de cálculo (*if, case, for, while, repeat*) e nos pesos associados às condições, atribuições e chamadas a pseudo-procedimentos.

5.4.3 Exemplo de Aplicação do ACME

Na FIGURA 5.5 é mostrado um exemplo de algoritmo para o ACME e na FIGURA 5.6 encontra-se o resultado de sua análise.

```
Program Quicksort;
var dados: array [1 .. n] of tipoqualquer;
   m,n,j:integer;
   pilharecursao:pilha;
begin
  pilharecursao:=nil;
  empilha(pilharecursao,1,n) #5;
  while topo(pilharecursao)<>nil do
    begin
      m:=topo(pilharecursao,m);
      n:=topo(pilharecursao,n);
      desempilha(pilharecursao) #6;
      while (n > m) do
        begin
          if dados[m+1] > dados[m]
            then
              swap(dados[m+1],dados[m]) #3
```

```
          else;
         j:=particao(m,n);
         if (j-m) > (n-j)
          then
          begin
            empilha(pilharecursao,m,j-1) #5;
            m:=j+1;
          end
          else
          begin
            empilha(pilharecursao,j+1,n) #5;
            n:=j-1;
         end;
       end;
   end;
end.
```

FIGURA 5.5 - Algoritmo Quicksort a ser analisado pelo ACME

X2'1 = C(pilharecursao := nil) = 0

X2'2 = C(empilha(pilharecursao,1,n)) = 5

X4'1 = C(m := topo(pilharecursao,m)) = 0

X4'2 = C(n := topo(pilharecursao,n)) = 0

X4'3 = C(desempilha(pilharecursao)) = 6

X7'1 = C(swap(dados[m+1],dados[m])) = 3

X7'2 = C(Comando VAZIO) = 0

Xq1 = C(dados[m+1]>dados[m]) = 0

P1 = P(dados[m+1]>dados[m])

X6'1 = C(If dados[m+1]>dados[m] then S7'1 else S7'2) = [Xq1 + P1.X7'1 + (1-P1).X7'2]

X6'1 = [0 + p(dados[m+1]>dados[m]).(3) + (1-p(dados[m+1]>dados[m])).(0)]

X6'2 = C(j := particao(m,n)) = 0

X8'1 = C(empilha(pilharecursao,m,j-1)) = 5

X8'2 = C(m := j+1) = 0

X7'3 = C(Begin S8'1 ... S8'2 End) = [SOM(i=1,2,X8'i)]

X7'3 = [5 + 0]

X8'3 = C(empilha(pilharecursao,j+1,n)) = 5

X8'4 = C(n := j-1) = 0

X7'4 = C(Begin S8'3 ... S8'4 End) = [SOM(i=3,4,X8'i)]

X7'4 = [5 + 0]

Xq2 = C((j-m)>(n-j)) = 0

P2 = P((j-m)>(n-j))

X6'3 = C(If (j-m)>(n-j) then S7'3 else S7'4) = [Xq2 + P2.X7'3 + (1-P2).X7'4]

X6'3 = [0 + p((j-m)>(n-j)).([5 + 0]) + (1-p((j-m)>(n-j))).([5 + 0])]

X5'1 = C(Begin S6'1 ... S6'3 End) = [SOM(i=1,3,X6'i)]

X5'1 = [[0 + p(dados[m+1]>dados[m]).(3) + (1-p(dados[m+1]>dados[m])).(0)] + 0 + [0 + p((j-m)>(n-j)).([5 + 0]) + (1-p((j-m)>(n-j))).([5 + 0])]]

Xq3 = C((n>m)) = 0

P3 = P((n>m))

X4'4 = C(While (n>m) do S5'1) = [Xq3+(X5'1+Xq3).P3/(1-P3)]

X4'4 = [0+([[0 + p(dados[m+1]>dados[m]).(3) + (1-p(dados[m+1]>dados[m])).(0)] + 0 + [0 + p((j-m)>(n-j)).([5 + 0]) + (1-p((j-m)>(n-j))).([5 + 0])]]+0).p((n>m))/(1-p((n>m)))]

X3'1 = C(Begin S4'1 ... S4'4 End) = [SOM(i=1,4,X4'i)]

X3'1 = [0 + 0 + 6 + [0+([[0 + p(dados[m+1]>dados[m]).(3) + (1-p(dados[m+1]>dados[m])).(0)] + 0 + [0 + p((j-m)>(n-j)).([5 + 0]) + (1-p((j-m)>(n-j))).([5 + 0])]]+0).p((n>m))/(1-p((n>m)))]]

Xq4 = C(topo(pilharecursao)<>nil) = 0

P4 = P(topo(pilharecursao)<>nil)

X2'3 = C(While topo(pilharecursao)<>nil do S3'1) = [Xq4+(X3'1+Xq4).P4/(1-P4)]

X2'3 = [0+([0 + 0 + 6 + [0+([[0 + p(dados[m+1]>dados[m]).(3) + (1-p(dados[m+1]>dados[m])).(0)] + 0 + [0 + p((j-m)>(n-j)).([5 + 0]) + (1-p((j-m)>(n-j))).([5 + 0])]]+0).p((n>m))/(1-p((n>m)))]]+0).p(topo(pilharecursao)<>nil)/(1-p(topo(pilharecursao)<>nil))]

X1 = C(Begin S2'1 ... S2'3 End) = [SOM(i=1,3,X2'i)]

X1 = SOM(i=1,3,X2'i)

X1 = [0 + 5 + [0+([0 + 0 + 6 + [0+([[0 + p(dados[m+1]>dados[m]).(3) + (1-p(dados[m+1]>dados[m])).(0)] + 0 + [0 + p((j-m)>(n-j)).([5 + 0]) + (1-p((j-m)>(n-j))).([5 + 0])]]+0).p((n>m))/(1-p((n>m)))]]+0).p(topo(pilharecursao)<>nil)/(1-p(topo(pilharecursao)<>nil))]]

FIGURA 5.6 - Desenvolvimento do cálculo do *Quicksort* feito pelo ACME

A notação p(x) denota a probabilidade da condição x ser verdadeira.

A complexidade média é medida, também, em função das probabilidades que nesse sistema ficam indicadas, pois essas probabilidades não são características do algoritmo, mas da distribuição dos dados. Esse sistema não supõe uma distribuição uniforme ou qualquer outras distribuição dos dados. Isto fica a cargo do usuário.

A variável X1 irá corresponder à equação de complexidade correspondente ao algoritmo *Quicksort*.

A saída do sistema é uma expressão bastante complicada, principalmente por querer ser geral e deixar a probabilidade de cada ramo de execução como um parâmetro a ser definido pelo usuário.

5.5 Comparativo do Sistema ANAC com os demais Sistemas

Neste capítulo, em suas seções anteriores, foram apresentados alguns dos principais sistemas desenvolvidos com a finalidade de analisar, de maneira automática, algoritmos. Estes sistemas possuem o grande mérito de sua inovação e pioneirismo no estudo e desenvolvimento de sistemas nesta área e com esta finalidade. Entretanto, esses sistemas apresentam alguns aspectos limitantes que procurou-se solucionar com este trabalho e o desenvolvimento do sistema ANAC.

O sistema METRIC foi desenvolvido na década de setenta, os sistema ACE e Lambda-Upsilon-Omega, foram desenvolvidos na década de oitenta. Entretanto, não encontra-se na literatura muitas referências a sistemas que tenham sido desenvolvidos com base nestes outros sistemas, nem muitos exemplos de aplicações da utilizações destes sistemas. Este fato pode vir a depor a favor da dificuldade da utilização, desenvolvimento de programas, ou ainda, dificuldade na análise dos resultados gerados por estes sistemas.

O sistema METRIC utiliza as medidas de complexidade apresentadas em [KNU 68], que são a complexidade no melhor caso, pior caso e caso médio, ao passo que o sistema ANAC realiza a análise apenas no pior caso. Entretanto, as medidas apresentadas no METRIC são voltadas para uma microanálise de programas, o que torna o sistema METRIC mais preciso do que o sistema ANAC, uma vez que este realiza macroanálise de programas, porém, pode-se observar na seção 2.3, que o comportamento assintótico de um algoritmo é o mais procurado, já que para um volume grande de dados é que a complexidade torna-se mais importante. O algoritmo assintoticamente mais eficiente é melhor para todas entradas, exceto entradas relativamente pequenas. Além disso, a microanálise de programas utilizada pelo METRIC pode gerar fórmulas intratáveis e de pouca utilidade prática para o analista. Por isso, a análise assintótica realizada pelo ANAC é mais interessante para a análise de algoritmos que irão operar com grande volume de dados, além disso as equações apresentadas pelo sistema ANAC é, sempre que possível, simplificada.

Outro aspecto é que o sistema METRIC manipula algoritmos desenvolvidos na linguagem Lisp, o que o torna bem específico no seu uso. O ANAC é mais geral no seu uso, por utilizar uma linguagem Pascal-like e não se prender ao tipo de problema que o algoritmo está buscando resolver e sim concentrar-se nas estruturas algorítmicas que compõem o algoritmo.

O sistema ACE foi um sistema desenvolvido sob forte influência do sistema METRIC. No entanto, o sistema ACE analisa programas no pior caso e ocupa-se com a macroanálise de programas. Sob esse aspecto muito se assemelha ao sistema ANAC, mas o sistema analisa algoritmos escritos em uma versão de programas funcionais, apresentada em [BAC 78] diferentemente do sistema ANAC que analisa algoritmos não

recursivos escritos numa linguagem imperativa. Entretanto, a primeira linguagem de programação aprendida por um aluno de computação é geralmente uma linguagem imperativa e Pascal-like, por isso a escolha no sistema ANAC de orientá-lo para esse tipo de linguagem.

Em relação ao sistema Lambda-Upsilon-Omega, pode-se afirmar que este apresenta, certamente, o tipo de análise algorítmica mais satisfatória dentre os sistemas anteriormente citados (METRIC, ACE e ACME), por gerar resultaodos mais fáceis de serem analisados. Entretanto, pode-se observar no seu exemplo de aplicação que o conhecimento para utilizar esta ferramenta deve abranger além da linguagem funcional Adl, o sistema operacional Unix, que é a plataforma na qual este sistema opera, e o software matemático Maple, que é utilizado para o processo de simplificação das equações geradas pelo sistema Lambda-Upsilon-Omega. Sem um conhecimento prévio destes ambientes e plataformas o uso do sistema Lambda-Upsilon-Omega fica extremamente comprometido. Pode-se, ainda, observar que o resultado gerado em ordem assintótica de complexidade pelo sistema ANAC é mais simples de ser analisado do que o resultado gerado pelo sistema Lambda-Upsilon-Omega, e um dos objetivos principais deste trabalho é apoiar o ensino de complexidade de algoritmos, tornando o seu estudo e conseqüentemente o cálculo de complexidade de algoritmos, algo mais simples de ser realizado e analisado, fazendo desta forma com que os usuários (alunos) possam fazer disto uma prática constante o que irá beneficiar o desenvolvimento de algoritmos eficientes.

O sistema ACME é o mais similar ao sistema ANAC, por utilizar a mesma metodologia empregada neste sistema e por trabalhar com algoritmos não recursivos escritos na linguagem Pascal-like. Embora a metodologia aplicada no ACME seja para análise no caso médio, a complexidade é calculada em função das estruturas algorítmicas, sem levar em conta o problema que o algoritmo está tentanto solucionar. Porém, no exemplo apresentado na seção 5.4.3, pode-se ver que o resultado gerado pelo sistema ACME é uma equação de complexidade e, na maioria das vezes, bastante difícil de ser analisada, por não gerar esta equação de forma simplificada. No sistema ANAC, como já foi dito anteriormente, além de simplificar esta equação o resultado é dado em ordem assintótica de complexidade.

Um aspecto importante do sistema ANAC é permitir o uso de subrotinas ausentes, o que permite um estudo modularizado do algoritmo e a utilização de rotinas prontas com complexidades conhecidas. O ANAC possui uma Tabela de complexidade com algumas das principais rotinas do estudo de algoritmos, tais como "classifica", "busca binária", "busca seqüêncial", dentre outras. Durante o processo de análise do algoritmo, ao encontrar chamadas para essas rotinas o ANAC considera a complexidade tabulada para a rotina encontrada. O usuário pode acrescentar ou modificar a Tabela para seu uso particular. Essa particularidade do ANAC permite que algoritmos sejam analisados e sua analise possa ser inserida na Tabela para que este algoritmo possa ser utilizado em outros algoritmos. Isto permite que o usuário (aluno) possa fazer vários experimentos e obter o resultado quase que instantaneamente, agilizando muito o processo de aprendizagem.

Nos sistemas anteriormente citados, METRIC, ACE, ACME e Lambda-Upsilon-Omega, essa facilidade não existe. Nestes sistemas, o usuário teria que acrescentar no algoritmo analisado todas as rotinas com o seu bloco de instruções para que a análise pudesse ser executada.

Outras vantagens que podem ser apontadas do ANAC em relação aos demais sistemas são:

O método utilizado pelo ANAC, faz com que a análise de algoritmos possa ser realizada independente de problemas ou classes;

Os fatores que influem na complexidade do algoritmo são facilmente identificáveis dentro das equações resultantes de complexidade, o que facilita a detecção de pontos ótimos e críticos dentro do algoritmo;

A simbologia utilizada no desenvolvimento do cálculo é sempre a mesma, de forma a não confundir o projetista ou analista;

O projetista de algoritmos pode realizar mudanças em seu algoritmo e analisar o efeito desta mudança na complexidade quase que instantaneamente, sem a necessidade de refazer cálculos;

O ANAC realiza avaliação matemática sobre as equações de complexidade, gerando a resolução final e resumida, sem a necessidade de conhecer softwares matemáticos para a simplificação das equações;

A interface com o usuário é totalmente gráfica e bastante simples, tornado seu uso bastante fácil. O sistema apresenta-se como um editor de texto, as principais funções existentes em qualquer editor de texto;

Permite o uso de subrotinas ausentes, onde o usuário especifica somente sua função de complexidade;

Possui uma metodologia de cálculo de complexidade bem clara, que o usuário pode acompanhar e facilmente realizar experimentos, como, por exemplo, trocar uma rotina e verificar o efeito que isto causa na complexidade final do algoritmo.

6 Conclusões

A apresentação em [TOS 90] da análise da complexidade de algoritmos como uma tarefa sistemática a partir de estruturas algorítmicas, e não como é classicamente apresentada como uma tarefa particular para cada algoritmo, tornou possível automatizar o cálculo de complexidade de algoritmos.

Durante as últimas décadas, poucos, mas significativos, sistemas surgiram com a finalidade de automatizar o processo de análise. Alguns dos principais sistemas são: o METRIC [WEG 75], o ACE [MET 88] e o Lambda-Upsilon-Omega [FLA 88]. Todas essas ferramentas analisam algoritmos especificados em uma determinada linguagem, todas funcionais, exceto o sistema ACME [SIL 97]. Os sistemas METRIC, Lambda-Upsilon-Omega e ACE, analisam somente algoritmos recursivos, gerando a equação de recorrência e resolvendo o primeiro através de equações de diferenças, como [LOR 00], o segundo através de funções geradoras e o terceiro através de indução estrutural. No sistema ANAC, optou-se por analisar algoritmos não recursivos escritos numa linguagem Pascal-like, o que o torna mais utilizável, por ser o Pascal uma linguagem amplamente difundida no meio acadêmico, ao contrário de outros sistemas que trabalham com a linguagem Lisp ou outras linguagens cuja utilização é mais restrita.

O sistema ANAC é uma ferramenta de apoio ao ensino de complexidade de algoritmos não recursivos, constituindo-se num ambiente interativo entre o usuário e o sistema durante o processo de análise. É um sistema simples de ser usado, que não exige conhecimento de um sistema operacional específico, software matemático ou linguagens pouco conhecidas, como o Lambda-Upsilon-Omega onde o usuário necessita ter conhecimento do sistema operacional Unix, conhecer a linguagem de programação Lisp e ainda ter conhecimento do software matemático Maple.

É um sistema interativo que vai guiando o usuário durante o processo de análise, isto faz com que o usuário vá adquirindo perícia no processo de análise e venha a tornar o processo de análise um hábito durante o processo de desenvolvimento de um algoritmo.

O sistema ANAC permite o uso de subrotinas que não estejam presentes no corpo do algoritmo, ou seja, existe apenas a chamada a uma rotina sem a necessidade de definir instruções em seu corpo. Para cumprir esta finalidade o sistema conta com uma Tabela de subrotinas definidas. Nesta Tabela estão presentes algumas rotinas bastante comuns em complexidade de algoritmos, tais como "classifica", "busca binária", dentre outras. O usuário pode incluir novas rotinas, inserindo na Tabela o nome da rotina e a complexidade da rotina. Existe a possibilidade de alterar esta Tabela, desta forma o usuário pode fazer vários experimentos no algoritmo e obter o resultado de maneira bastante rápida.

O resultado da análise realizada pelo sistema é simplificado, dado em ordem de complexidade, diferente de outros sistemas que podem gerar extensas e complicadas equações, que podem ser desestimulantes para quem não esteja habituado com o processo de cálculo de complexidade ou para quem está se iniciando no estudo da Teoria da Complexidade.

O sistema ANAC pretende tornar a análise de algoritmos uma prática freqüente no projeto e desenvolvimento de algoritmos eficientes. A utilização do sistema ou método deverá criar no projetista (usuário) uma cultura de desenvolvimento em que esta prática acabe se tornando uma tarefa bem menos árdua do que pode-se ver atualmente em equipes ou profissionais que desenvolvem algoritmos e buscam eficiência e, no entanto, fogem à prática da análise da complexidade de algoritmos.

Destina-se a alunos dos primeiros semestres de cursos de computação e sistemas de informação, analisando algoritmos escritos numa linguagem simples, Pascal-Like. Usa uma metodologia de cálculo bem determinada (descrita no capítulo 2) e tem facilidade para se executar experimentos, tais como substituir rotinas pré-definidas. Pode tratar, uma parte do algoritmo cuja complexidade já lhe é conhecida, como um módulo, substituindo-o por uma rotina, especificando sua complexidade e podendo, desta forma, analisar outras modificações no restante do algoritmo.

A contradição entre a preocupação com a eficiência dos algoritmos e a não preocupação em calcular a complexidade deles é uma situação que deve ser revertida. Assim, um resultado que almeja-se alcançar com esse trabalho é o despertar da necessidade para essa importante tarefa no desenvolvimento de algoritmos, fazendo do cálculo da complexidade uma etapa do projeto do algoritmo, e então aumentar o número de algoritmos analisados e a geração de algoritmos mais eficientes.

Anexo 1 Gramática da Linguagem Pascal-Like Utilizada na Descrição dos Algoritmos (NOTAÇÃO BNF)

```
<prog> ::= program <ident>; begin <sequencia> end.

<sequencia> ::= <comando> ;

<comando>  ::=   <sequencia>    |
                 <atribuicao>   |
                 <if>           |
                 <for>          |
                 <while>

<atribuicao>  ::= <ident> := <expressao>

<if> ::= if <condicao> then <comando> else <comando> endif
         |  if <condicao> then <comando> endif

<for> ::= for<atribuicao> to <expressao> do <comando>endfor

<while> ::= while <condicao> do <comando> endwhile

<condicao>  ::= <expressao> <relacional> <expressao> |
                <condicao> and <condicao> |
                <condicao> or <condicao>  |
                not <condicao> ( <condicao> ) <expressao>

<expressao> ::= <expressao> <operador> <expressao> |
                - <expressao> | ( <expressao> ) |
                <ident> [( parametros )] |
                <variavel> |
                numero | true | false | string
```

<relacional> ::= <|> | <> | <= |>= | =

<operador> ::= + | - | * | /

<parametros> ::= <parametros> [, <parametros>] |
<expressao> | *string*

<ident> ::= *identificador*

Anexo 2 Exemplos de Aplicação do ANAC

Este anexo apresenta mais exemplos de aplicação da ferramenta ANAC. Os algoritmos usados como exemplo foram retirados de [ORT 01].

```
program raizes;
begin
        read ('a,b,c'  );
        if a = 0  then
                if b <> 0 then
                        x := -c/b;
                        write ('  x'  );
                   else
                        write ('  c'  );
                endif;
           else
                 d := b*b-4*a*c;
        endif;
        if d < 0 then
                 write ('Equacao sem raizes reais'  );
          else
                x1 := (-b +sqrt(d))/(2*a);
                x2 := (-b - srqt(d))/(2*a);
                write (' x  1,x2'  );
        endif;
end.
```

Pior Caso

Analisando o algoritmo...

```
x1 = C ( read ( a,b,c ) + x2 =  1  + x2
x2 = C ( a=0 ) + x3 + x4
x3 = C ( b<>0 ) + x5 + x6
x5 = C ( x := -c / b ) + x7 =  1  + x7
x7 = C ( write ( x ) + x8 =  1  + x8
x8 = C ( write ( c ) + x9 =  1  + x9
x9 = C (  endif  )  = 0
x8 = 1 + 0 = 1
x7 = 1 + 1 = 1
x5 = 1 + 1 = 1
x6 = C ( d := b*b-4*a*c ) + x10 =  1  + x10
x10 = C ( endif )  = 0
x6 = 1 + 0 = 1
x3 = 1 + 1 + 1 = 1
x4 = C ( d<0 ) + x11 + x12
x11 = C ( write ( Equacao sem raizes reais ) + x13 =  1  + x13
x13 = C ( x1 := (-b + sqrt(d))/(2*a)) + x14 =  1  + x14
x14 = C ( x2 := (-b -  sqrt(d))/(2*a)) + x15 =  1  + x15
x15 = C ( write (  x1,x2 ) + x16 =  1  + x16
```

x16 = C (endif) = 0
x15 = 1 + 0 = 1
x14 = 1 + 1 = 1
x13 = 1 + 1 = 1
x11 = 1 + 1 = 1
x4 = 1 + 1 + 1 = 1
x2 = 1 + 1 + 1 = 1
x1 = 1 + 1 = 1

O(1)

```
program media_geral;
begin
        soma := 0;
        cont  := 0;
        read ('n  um, n1,n2,n3'  );
        while num > 0 do
                media := (n1+n2+n3)/3;
                soma := soma + media;
                cont := cont + 1;
                read ('n  um, n1,n2,n3'  );
        endwhile;
        medgeral := soma/cont;
        write ('medgeral'  );
end.
```

Pior Caso

Analisando o algoritmo...

x1 = C (soma := 0) + x2 = 1 + x2
x2 = C (cont := 0) + x3 = 1 + x3
x3 = C (read (num, n1,n2,n3) + x4 = 1 + x4
x4 = C (while num > 0 do x5) + x6 = C (num>0) + [n.x5] + x6
x5 = C (media := (n1+n2+n3)/3) + x7 = 1 + x7
x7 = C (soma := soma + media) + x8 = 1 + x8
x8 = C (cont := cont + 1) + x9 = 1 + x9
x9 = C (read (num, n1,n2,n3) + x10 = 1 + x10
x10 = C (endwhile) = 0
x9 = 1 + 0 = 1
x8 = 1 + 1 = 1
x7 = 1 + 1 = 1
x5 = 1 + 1 = 1
x6 = C (medgeral := soma / cont) + x11 = 1 + x11
x11 = C (write (num, n1,n2,n3) + x12 = 1 + x12
x12 = C (end) = 0
x11 = 1 + 0 = 1
x6 = 1 + 1 = 1
x4 = 1 + [n.1] + 1 = n
x3 = 1 + n = n

x2 = 1 + n = n
x1 = 1 + n = n

O (n)

```
program pesquisa_em_TABELA;
begin
        for i := 1 to n do
                for j := 1 to n do
                        read ('  T[j]'  );
                endfor;
        endfor;
        for z := 1 to  n do
                read ('  a'  );
                for i := 1 to n do
                        for j := 1 to n do
                                if  a = t[i,j] then
                                        i := 12;
                                        j := 10;
                                endif;
                        endfor;
                endfor;
        if i = 11 then
                write ('  a não esta na tabela'  );
        else
                write ('  a  esta na tabela'  );
        endif;
        endfor;
end.
```

Pior Caso

Analisando o algoritmo...

x1 = C (for i := 1 to n do x2) + x3 = [SUM(i = 1 , n) x2] + x3
x2 = C (for j := 1 to n do x4) + x5 = [SUM(j = 1 , n) x4] + x5
x4 = C (read (T[i,j]) + x6 = 1 + x6
x6 = C (endfor) = 0
x4 = 1 + 0 = 1
x5 = C (endfor) = 0
x2 = [SUM (j = 1 , n) 1] + 0 = n + 0 = n
x3 = C (for z := 1 to n do x7) + x8 = [SUM(z = 1 , n) x7] + x8
x7 = C (read (a) + x9 = 1 + x9
x9 = C (for i := 1 to n do x10) + x11 = [SUM(i = 1 , n) x10] + x11
x10 = C (for j := 1 to n do x12) + x13 = [SUM(j = 1 , n) x12] + x13
x12 = C (a=t[i,j]) + x14 + x15
x14 = C (i := 12) + x16 = 1 + x16
x16 = C (j := 10) + x17 = 1 + x17
x17 = C (endif) = 0
x16 = 1 + 0 = 1

x14 = 1 + 1 = 1
x15 = C (endfor) = 0
x12 = 1 + 1 + 0 = 1
x13 = C (endfor) = 0
x10 = [SUM (j = 1 , n) 1] + 0 = n + 0 = n
x11 = C (i=11) + x18 + x19
x18 = C (write (a não esta na tabela) + x20 = 1 + x20
x20 = C (write (a esta na tabela) + x21 = 1 + x21
x21 = C (endif) = 0
x20 = 1 + 0 = 1
x18 = 1 + 1 = 1
x19 = C (endfor) = 0
x11 = 1 + 1 + 0 = 1
x9 = [SUM (i = 1 , n) n] + 1 = n^2+1 = n^2
x7 = 1 + n^2 = n^2
x8 = C (end) = 0
x3 = [SUM (z = 1 , n) n^2] + 0 = n^3+0 = n^3
x1 = [SUM (i = 1 , n) n] + n^3 = n^2+n^3 = n^3

O(n^3)

```
program fatorial;
begin
        read ('  n'  );
        fat := 1;
        for i := 2 to n do
                fat := fat * i;
        endfor;
        fatorial := fat;
end.
```

Pior Caso

Analisando o algoritmo...

x1 = C (read (n) + x2 = 1 + x2
x2 = C (fat := 1) + x3 = 1 + x3
x3 = C (for i := 2 to n do x4) + x5 = [SUM(i = 2 , n) x4] + x5
x4 = C (fat := fat * i) + x6 = 1 + x6
x6 = C (endfor) = 0
x4 = 1 + 0 = 1
x5 = C (fatorial := fat) + x7 = 1 + x7
x7 = C (end) = 0
x5 = 1 + 0 = 1
x3 = [SUM (i = 2 , n) 1] + 1 = n + 1 = n
x2 = 1 + n = n
x1 = 1 + n = n

O (n)

```
program MaiorLinha;
begin
        maior := Mat[1,1];
        for i := 2 to n do
                if maior < Mat[1,i] then
                        maior := Mat[1,i];
                endif;
        endfor;
        MaiorLinha := maior;
end.
```

Pior Caso

Analisando o algoritmo...

x1 = C (maior := Mat[1,1]) + x2 = 1 + x2
x2 = C (for i := 2 to n do x3) + x4 = [SUM(i = 2 , n) x3] + x4
x3 = C (maior<Mat[1,i]) + x5 + x6
x5 = C (maior := Mat[1,i]) + x7 = 1 + x7
x7 = C (endif) = 0
x5 = 1 + 0 = 1
x6 = C (endfor) = 0
x3 = 1 + 1 + 0 = 1
x4 = C (MaiorLinha := maior) + x8 = 1 + x8
x8 = C (end) = 0
x4 = 1 + 0 = 1
x2 = [SUM (i = 2 , n) 1] + 1 = n + 1 = n
x1 = 1 + n = n

O (n)

```
program EscrevaMat;
begin
        for i:=1 to n do
                for i:= 1 to n do
                        write ('Mat[i,j]' );
                endfor;
        endfor;
end.
```

Pior Caso

Analisando o algoritmo...

x1 = C (for i := 1 to n do x2) + x3 = [SUM(i = 1 , n) x2] + x3
x2 = C (for i := 1 to n do x4) + x5 = [SUM(i = 1 , n) x4] + x5
x4 = C (write (Mat[i,j]) + x6 = 1 + x6
x6 = C (endfor) = 0
x4 = 1 + 0 = 1
x5 = C (endfor) = 0

x2 = [SUM (i = 1 , n) 1] + 0 = n + 0 = n
x3 = C (end) = 0
x1 = [SUM (i = 1 , n) n] + 0 = n^2+0 = n^2

O(n^2)

```
program LeiaMat;
begin
        for i:=1 to n do
                for i:= 1 to n do
                        read ('Mat[i,j]' );
                endfor;
        endfor;
end.
```

Pior Caso

Analisando o algoritmo...

x1 = C (for i := 1 to n do x2) + x3 = [SUM(i = 1 , n) x2] + x3
x2 = C (for i := 1 to n do x4) + x5 = [SUM(i = 1 , n) x4] + x5
x4 = C (read (Mat[i,j]) + x6 = 1 + x6
x6 = C (endfor) = 0
x4 = 1 + 0 = 1
x5 = C (endfor) = 0
x2 = [SUM (i = 1 , n) 1] + 0 = n + 0 = n
x3 = C (end) = 0
x1 = [SUM (i = 1 , n) n] + 0 = n^2+0 = n^2

O(n^2)

Anexo 3 Artigos Publicados

Anexo 3.1 SBIE 2000

Ferramenta para Automatização da Análise da Complexidade de Algoritmos

Artigo Completo

Marco Antonio de Castro Barbosa
Laira Vieira Toscani
Leila Ribeiro

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Instituto de Informática
Departamento de Informática Teórica
Av. Bento Gonçalvez, 9500 Blc. IV. Agronomia
Porto Alegre – RS
91501-970 – Caixa Postal: 15064
Tel (51) 316-6845

{mantonio,laira,leila}@inf.ufrgs.br

Resumo:

Este trabalho apresenta a concepção, projeto e desenvolvimento do protótipo de sistema ANAC que consiste de uma ferramenta a ser utilizada no apoio ao ensino de complexidade de algoritmos. O sistema tem por principal objetivo guiar o usuário no processo de análise do algoritmo fornecendo como resultado as equações de complexidade para o caso médio e pior caso.

Palavras-chave: complexidade de algoritmos, análise de algoritmos, pior caso, caso médio, análise automática, apoio ao ensino.

BARBOSA, M. A. C., TOSCANI, L. V., RIBEIRO, L. Ferramenta para Automatização da Análise da Complexidade de Algoritmos In: XI - Simpósio Brasileiro de Informática na Educação, 2000, Maceió.

Anexo 3.2 SIIE 2001

ANAC – Uma Ferramenta para Apoio ao Ensino de Complexidade de Algoritmos

Marco Antonio de Castro Barbosa
Laira Vieira Toscani
Leila Ribeiro

Universidade Federal do Rio Grande do Sul
Instituto de Informática
Departamento de Informática Teórica
Porto Alegre-RS, Brasil, CEP 91.501-970

{mantonio,laira,leila}@inf.ufrgs.br

ABSTRACT

This paper presents the conception, project and development of the prototype system ANAC that consists of a tool to be used in aid to teaching complexity of algorithms. The system has for main objective to guide the user in the process of analysis of algorithm supplying as result the complexity equations for the average case and worst case.
Key-words: complexity of algorithms, analysis of algorithms, worst case, average case, automatic analysis, aid to teaching.

RESUMO

Este artigo apresenta a concepção, projeto e desenvolvimento do protótipo de sistema ANAC que consiste de uma ferramenta a ser utilizada no apoio ao ensino de Complexidade de Algoritmos. O sistema tem por principal objetivo guiar o usuário no processo de análise do algoritmo fornecendo como resultado as equações de complexidade para o pior caso ou caso médio.

Palavras Chaves: Complexidade de algoritmos, análise de algoritmos, apoio ao ensino.

BARBOSA, M. A. C., TOSCANI, L. V., RIBEIRO, L. ANAC - Uma Ferramenta para o Apoio ao Ensino de Complexidade de Algoritmos In: 3º Simpósio Internacional de Informática Educativa, 2001, Viseu.

Anexo 3.3 CLEI 2001

ANAC – Uma Ferramenta para Análise Automática da Complexidade de Algoritmos

Marco Antonio de Castro Barbosa
Laira Vieira Toscani
Leila Ribeiro
Universidade Federal do Rio Grande do Sul
Instituto de Informática
Departamento de Informática Teórica
Porto Alegre-RS, Brasil, CEP 91.501-970
{mantonio,laira,leila}@inf.ufrgs.br

ABSTRACT

This paper presents the conception, project and development of the prototype system ANAC that consists of a tool to be used in aid to teaching complexity of algorithms. The system has for main objective to guide the user in the process of analysis of algorithm supplying as result the complexity equations for the average case and worst case.
Key-words: complexity of algorithms, analysis of algorithms, worst case, average case, automatic analysis, aid to teaching.

RESUMO

Este artigo apresenta a concepção, projeto e desenvolvimento do protótipo de sistema ANAC que consiste de uma ferramenta a ser utilizada no apoio ao ensino de Complexidade de Algoritmos. O sistema tem por principal objetivo guiar o usuário no processo de análise do algoritmo fornecendo como resultado as equações de complexidade para o pior caso ou caso médio.
Palavras Chaves: Complexidade de algoritmos, análise de algoritmos, apoio ao ensino.

BARBOSA, M. A. C., TOSCANI, L. V., RIBEIRO, L. ANAC - Uma Ferramenta para Análise Automática da Complexidade de Algoritmos In: XXVII Latin-American Conference on Informatics - CLEI' 2001, 2001, Merida.

Anexo 3.4 Revista do CCEI

Artigo publicado na Revista do CCEI. Universidade da Região da Campanha – Centro de Ciências da Economia e Informática. Volume 5 – Número 8, agosto de 2001.

ANAC – Uma Ferramenta para Análise Automática da Complexidade de Algoritmos

Marco Antonio de Castro Barbosa

Laira Vieira Toscani

Leila Ribeiro

RESUMO

Este artigo apresenta a concepção, projeto e desenvolvimento do protótipo de sistema ANAC que consiste de uma ferramenta a ser utilizada no apoio ao ensino de Complexidade de Algoritmos. O sistema tem por principal objetivo guiar o usuário no processo de análise do algoritmo fornecendo como resultado as equações de complexidade para o pior caso ou caso médio.

ABSTRACT

This paper presents the conception, project and development of the prototype system ANAC that consists of a tool to be used in aid to teaching complexity of algorithms. The system has for main objective to guide the user in the process of analysis of algorithm supplying as result the complexity equations for the average case and worst case.

BARBOSA, M. A. C., TOSCANI, L. V., RIBEIRO, L. ANAC - Uma Ferramenta para Análise Automática da Complexidade de Algoritmos. Revista do CCEI. Urcamp, Bagé, 2001.

Bibliografia

[AHO 74] AHO, A.; HOPCROFT, J.; ULLMAN, J. **The Design and Analysis of Computer Algorithms**. Massachusetts: Addison-Wesley, 1974.

[AHO 82] AHO, A.; HOPCROFT, J.; ULLMAN, J. **Data Structures and Algorithms**. Massachusetts: Addison-Wesley, 1982.

[BAC 78] BACKUS, J. W. Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs. **Communications of ACM,** New York, v.21, n.8, p.613-641, Aug. 1978.

[BAC 81] BACKUS, J. W. **The algebra of functional programs**: function level reasoning, linear equations, and extended definitions. New York: Springer Verlag, 1981. p.1-43. (Lecture Notes in Computer Science, v. 27).

[BAR 2000] BARBOSA, Marco Antonio de Castro; TOSCANI, Laira Vieira; RIBEIRO, Leila. Ferramenta para automatização da análise da Complexidade de algoritmos. In: SIMPÓSIO BRASILEIRO DE INFORMÁTICA NA EDUCAÇÃO, SBIE,11., 2000, Maceió. **Anais...** Maceió: UFAL, 2000.

[BAR 2001a] BARBOSA, Marco Antonio de Castro; TOSCANI, Laira Vieira; RIBEIRO, Leila. ANAC - Ferramenta para o Apoio ao Ensino de Complexidade de Algoritmos. In: SIMPÓSIO INTERNACIONAL DE INFORMÁTICA EDUCATIVA, SIIE, 3., 2001, Viseu. **Anais...**Viseu: ESEV, 2001.

[BAR 2001b] BARBOSA, Marco Antonio de Castro; TOSCANI, Laira Vieira; RIBEIRO, Leila. ANAC – Uma ferramenta para análise automática da Complexidade de algoritmos. In: CLEI, 27., 2001, Mèrida. **Articulos**. Mèrida: Universidad de Los Andes, 2001.

[CHA 88] CHAR, B. W.; GEDDES, K. O.; GONNET, G. H. **MAPLE**: Reference Manual. Waterloo: University of Waterloo, 1988.

[COH 82] COHEN, J. Computer-assisted microanalysis of programs. **Communications of ACM**, New York, v.25, n.10, p.724-733, Oct. 1982.

[COO 83] COOK, S. An overview of computational complexity. **Communications of ACM**, New York, v.26, n.6, p.401-407, June 1983.

[COR 90] CORMEN, T. H.; LEISERSON, E. C.; RIVEST, R. L. **Introduction to Algorithms.** Cambridge: McGraw-Hill Books: The MIT Electrical Engineering and Computer Science Series, 1990.

[FLA 88] FLAJOLET, Philippe; SALVY, Bruno; ZIMMERMANN, Paul. ΛΥΩ: an assistante algorithms analyser. In: INTERNATIONAL CONFERENCE ON APPLICABLE ALGEBRA, ERROR - CORRETING CODES, COMBINATORES AND COMPUTER ALGEBRA, AAECC, 6., 1988. **Proceedings...** Berlim: Springer-Verlag, 1988.

[FLA 89] FLAJOLET, Philippe; SALVY, Bruno; ZIMMERMANN, Paul. **Lambda-Upsilon-Omega The 1989 Cookbook**. Le Chesnay Cedex: INRIA-Rocquencourt, 1989. (INRIA Reserch Report, 1073).

[GAR 79] GAREY, Michael R.; JOHNSON, David S. **Computers and Intractability – A Guide to the Theory of NP-Completeness**. San Francisco: W. H. Freeman and Company, 1979.

[HIC 88] HICKEY, T.; COHEN, J. Automating program analysis. **Journal of ACM**, New York: v.35, p.185-220, 1988.

[HOR 78] HOROWITZ, Ellis; SAHNI, Sartaj. **Fundamentals of Computer Algorithms**. Maryland: Computer Science Press, 1978.

[KAR 86] KARP, Richard M. Combinatorics, Complexity and Randomness. **Communications of ACM,** New York, v.29, n.2, p.98-109, Feb. 1986.

[KNU 68] KNUT, D. E. **The Art of Computer Programming**. Massachusetts: Addison-Wesley, 1968. v.1.

[KNU 73] KNUT, D. E. **The Art of Computer Programming**. Massachusetts: Addison-Wesley, 1973. v.3.

[KOZ 81] KOZEN, D. Semantics of probabilistic programs. **Journal of Computer System Sciences**, Los Angeles, v.22, p.328-350, 1981.

[LOR 99] LORETO, Aline B; TOSCANI, Laira Vieira; NECRÓN, Manuel M. Cálculo da Complexidade de Algoritmos Recursivos através de Equações Características. In: CONGRESSO NACIONAL DE MATEMÁTICA APLICADA E COMPUTACIONAL, CNMAC, 22., 1999, Santos. **Resumo das comunicações**. [S. l.: SBMAC], 1999.

[LOR 2000] LORETO, Aline B. **Cálculo da complexidade exata de algoritmos do tipo Divisão-e-conquista através das equações características**. Porto Alegre: PPGMAp-UFRGS, 2000.

[McC 60] McCARTHY, J. Recursive functions of symbolic expressions and their computation by machine. **Communications of ACM,** New York, v.3, n.4, p.184-185, Apr. 1960.

[McC 63] McCARTHY, J. A basis for a mathematical theory of computation. In: BRAFFORT, P.;HIRSBERG, D.(Ed.). **Computer Programming and Formal Systems**. Amsterdam: North-Holland, 1963.

[MCM 99] MCMANIS, Chuck. **Looking for lex and yacc for Java ? You don't know Jack**. Java World. 01 de julho de 1999. Disponível em: <http://www.javaworld.com/javaworld/jw-12-1996/jw-12-jack_p.html>. Acesso em: 10 jan. 2000.

[MEI 78] MEIER A.; MOON J. W. On the Altitude of Nodes in Random Trees. **Canadian Journal of Mathematics**, Ottawa, v.30, p.997-1015, 1978.

[MET 88] METAYER, D. Le. Ace: An automatic complexity evaluator. **ACM Transactions on Programming Languages and Systems**, New York, v.10, n.2, p.248-266, Apr. 1988.

[ORT 2001] ORTH, Afonso I. **Algoritmos e Programação**. Porto Alegre: AIO, 2001.

[PRI 2000] PRICE, Ana M de A.; TOSCANI, Simão S. **Implementação de Linguagens de Programação**: Compiladores. Porto Alegre: Sagra-Luzatto, 2001. (Livros Didáticos, n. 9).

[RAM 79] RAMSHAW, L. H. **Formalizing the analysis of algorithms**, 1979. Ph. D. Thesis, Standford University June 1979. Also available as Techinical Report. SL-79-5, Xerox Palo Alto Research Center, Palo Alto, California.

[ROS 97] ROSA, Débora Schuch da. **Complexidade Média Algoritmica:** uma metodologia para o seu cálculo. Porto alegre: CPGCC/UFRGS, 1997. Dissertação de Mestrado.

[SED 96] SEDGEWICK, Robert; FLAJOLET, Philippe. **An Introduction to the Analysis of Algorithms**. Massachusetts: Addison Wesley, 1996.

[SIL 98] SILVEIRA; Carlos Morelli D. **Analisador de Complexidade Média Baseado nas Estruturas Algorítmicas**. Pelotas: UFPEL, 1998.

[TAR 87] TARJAN, Robert E. Algorithm Design. **Communications of the ACM**, New York, v.30, n.3, p. 205-212, 1987.

[TER 91] TERADA, Routo. **Desenvolvimento de Algoritmos e Estruturas de Dados**. São Paulo: McGraw-Hill do Brasil, 1991.

[TOS 85] TOSCANI, L. V.; VELOSO, Paulo A. S. Uma especificação formal para a programação dinâmica. In: SEMINÁRIO INTEGRADO DE SOFTWARE E HARDWARE, 12., 1985, Porto Alegre. **Anais...** Porto Alegre: SBC, 1985. p.477-486

[TOS 90] TOSCANI, L. V.; VELOSO, Paulo A. S**.** Uma metodologia para cálculo da complexidade de algoritmos. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, SBC, 4., 1990, Águas de São Pedro: **Anais...**São Paulo: USP, 1990.

[TOS 2001] TOSCANI, L. V.; VELOSO, Paulo A. S. **Complexidade de Algoritmos**. Porto Alegre: Sagra-Luzzato, 2001. (Livros Didáticos, n. 13).

[VEL 81] VELOSO, Paulo A. S.; VELOSO, S. R. M. Problem Decomposition and Reduction: Applicability, Soundness, Completeness. In: PROGRESS in CYBERNETICS and SYSTEMS RESEARCH, 1981. **Proceedings...**Washington, D.C. Hemisphere Publishers Corp., 1981. p.199-203.

[WEG 75] WEGBREIT, B. Mechanical program analysis**. Communications of ACM**, New York, v.18, n.9, p.528-539, Jan. 1975.

[WEI 67] WEISSMAN, C. **Lisp 1.5 Primer**. Belmont, California: Dickenson Publication, 1967.

[WIL 82] WILLIAMS, J. On the development of the algebra of functional programs. **ACM Transactions of Programing Language System**, New York, v.4, n.4, p.733-757, Oct. 82.

[ZIM 88] ZIMMERMANN, W. **How to mechanize complexity analysis**. Karlsruhe: University of Karlsruhe, 1988. Technical report.