AVALIAÇÃO DA DISTRIBUIÇÃO DO TEMPO DE EXECUÇÃO EM PROGRAMAS

Fernando Antonio Vanini

Dissertação apresentada do Instituto de Matemática, Estatística e Ciência da Computação como requisito parcial para obtenção do título de Mestre em Ciência da Computação.

Campinas, junho de 1979.

RESUMO

Este trabalho descreve um sistema de avaliação da distribuição do tempo de execução em programas FORTRAN, desenvolvido na UNICAMP.

A avaliação é feita criando-se uma versão modificada do programa que ao ser executada determina o número de vezes que cada comando é executado. O tempo de execução de cada comando é estimado com base no número de execuções e num peso atribuído a cada comando. A contagem do número de execuções é feita inserindo contadores no programa. O número de contadores inseridos é mínimo e a minimização é feita com base em alguns resultados de teoria dos grafos.

Agradeço ao prof. Dr. Nelson de Castro Machado pela orientação e incentivo recebidos; ao prof. Dr. Claudio L. Lucchesi pela valiosa colaboração; ao prof. Dr. Jacques Cohen pela influência benéfica; a todos os colegas do IMECC que contribuiram para a conclusão deste trabalho.

CONTEÚDO

CAPÍTULO 1

Introdução

Os esforços no sentido de desenvolver métodos ou sistemas visando a otimização de programas caracterizam-se por adotar uma de duas políticas:

a -desenvolvimento de algoritmos que otimizam automaticamente o código gerado durante a compilação. Esta otimização é portanto invisível ao programador

b -desenvolvimento de ferramentas que permitam a obtenção de informações a respeito do programa, a fim deste ser posteriormente otimizado pelo próprio usuário. Sistemas desse tipo atuam monitorando a execução do programa a ser otimizado, ficando a função de otimização explicitamente a cargo do usuário.

Independente da política adotada para otimização de programas, um dos parâmetros mais importantes para a quantificação da eficiência de código é o tempo de execução, e como este se distribue pelas várias partes do programa. Obviamente outros parâmetros, como a quantidade de memória usada, utilização dos canais de entrada e saída, número de acessos à memória, etc. , devem ser considerados num tratamento mais abrangente do problema da otimização. Este trabalho, entretanto, focaliza o problema sob o ponto de vista da distribuição do tempo de execução entre os segmentos do programa.

Diversos métodos tem sido apresentados para avaliação da distribuição do tempo de execução. Os principais esquemas adotados são:

I - Avaliação por amostragem [3]. Uma rotina anexada ao programa interrompe-o periódicamente e incrementa um contador associado ao endereço da próxima instrução a ser executada. As instruções executadas com mais frequência terão maior probabilidade de ser interrompidas, logo seus endereços terão contagem mais alta. Finda a execução, é fornecida uma listagem de todos os endereços e o número de vezes que foram anotados. Tal método, além de ser impreciso e ineficiente, exige, para sua implementação a disponibilidade de recursos de sistema sofisticados e pouco comuns (interrupções controladas por programa de usuário). Na falta deste recurso, tornar-se-iam necessárias modificações no sistema operacional para a utilização deste método.

II- Inserção automática, no programa, de contadores que permitam a medição da frequência de execução de suas várias partes. Deste modo, o próprio programa, ao ser processado coleta informações sobre a execução. A listagem desses contadores fornecerá dados a partir dos quais o usuário otimizará o programa.

III-Análise formal do programa - o programa do usuário é dado como entrada para um sistema que o traduz para uma expressão algébrica representando o tempo de execução do programa em função

dos valores de entrada.

Este trabalho descreve a implementação de um sistema que se enquadra na segunda classe de métodos. No capítulo 2 é focalizado o problema da contagem do número de execuções dos comandos de um programa através da inserção de contadores. O capítulo 3 trata do problema de determinar um conjunto mínimo de contadores a serem introduzidos no programa. Nos capítulos 4 e 5 são feitas considerações sobre a implementação do sistema no computador DEC-10 da UNICAMP, e o capítulo 6 faz uma breve descrição sobre a forma de utilizar o sistema.

CAPÍTULO 2

Medição das Frequências de Execução dos Comandos

Descreveremos neste capítulo a técnica da medição da frequência de execução dos camandos através da inserção de contadores. Note-se que utilizamos neste trabalho, de maneira informal, o termo "frequência de execução" significando simplesmente a contagem do número de vezes que um dado comando é executado durante o processamento do programa.

A idéia trivial seria introduzir um contador para cada comando do programa. Essa técnica é de aplicação bastante simples, porém o tempo adicional gasto com seu uso pode ser grande em relação ao tempo de execução do programa na sua forma original.

Observe-se o trecho de programa P1, em FORTRAN, abaixo:

```
( 1)   3    IF ( I .EQ. J ) GO TO 1
( 2)        IF ( I .GT. J ) GO TO 2
( 3)        K = I
( 4)        I = J
( 5)        J = K
( 6)        GO TO 3
( 7)   2    I = I - J
( 8)        GO TO 3
( 9)   1    CONTINUE
```

Programa P1

Pode-se garantir que os comandos compreendidos entre as linhas (3) e (6), inclusive, serão executados um número igual de vezes pois são sempre executados sequencialmente. Isto implicã que os potenciais contadores para cada comando forneceriam informação redundante; um único contador para todo o trecho é suficiente.

Portanto, o número de pontos de medição no programa pode ser reduzido se localizarmos os trechos básicos (por trecho básico, entenda-se um trecho como o apresentado acima, cujos comandos são executados sequencialmente, tendo, portanto um único "ponto de entrada" e um único "ponto de saída") e em seguida inserirmos um contador para cada trecho básico do programa.

Além disso, a frequência de execução do comando na linha (1) é igual à soma das frequências dos comandos em (6) e (8) (supondo não haver outro desvio para o comando em (1)), portanto, se utilizássemos contadores em (1), (6) e (8), um deles poderia ser suprimido.

A fim de minimizar o número de contadores inseridos no programa, este será representado por um grafo no qual se localizará um conjunto mínimo de pontos de medição de fluxo, que corresponderá, no programa, ao conjunto dos trechos básicos onde serão inseridos contadores.

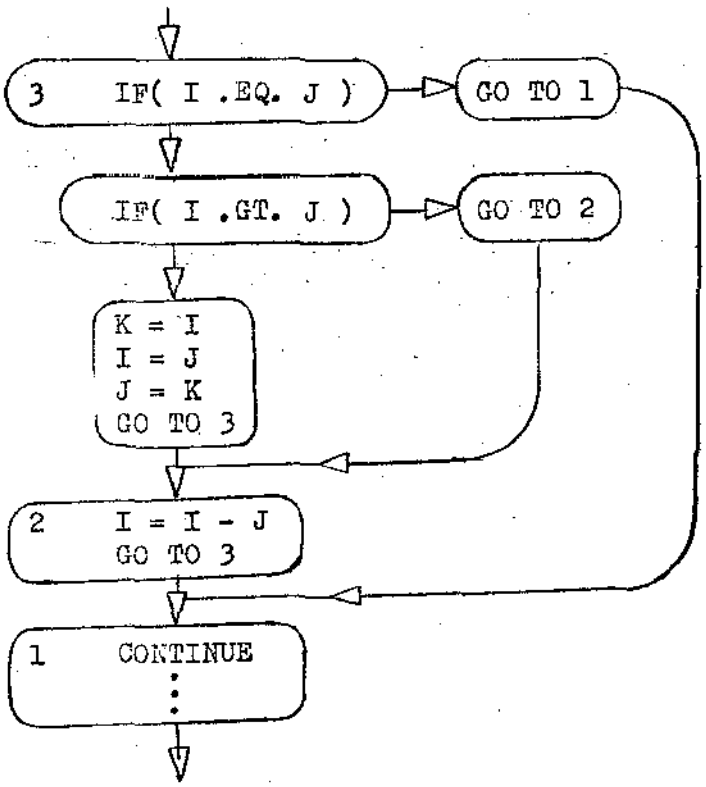
2.1 Construção do Grafo a Partir do Programa

A construção do grafo que irá representar um dado programa será feita da seguinte forma:

a - cada trecho básico do programa será representado por um vértice no grafo.

b - os desvios entre os trechos básicos serão representados pelas arestas do grafo.

Exemplo: o trecho de programa P1 seria representado pelo seguinte grafo

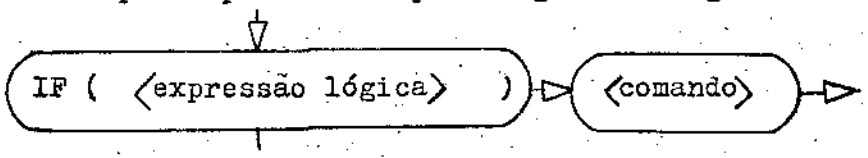


Observações:

1 - A fim de manter uma certa uniformidade e facilitar a implementação, o comando "if-lógico", da forma

IF (⟨expressão lógica⟩) ⟨comando⟩

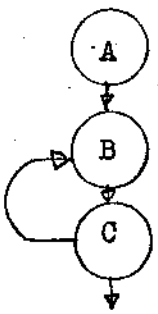
será sempre representado pelo seguinte subgrafo:



2 - O comando DO, da forma

DO ⟨n⟩⟨var⟩ = ⟨expr.1⟩, ⟨expr.2⟩, ⟨expr.3⟩
⟨sequência de comandos⟩
⟨n⟩ ⟨comando⟩

será representado por



onde

A - é o vértice que representa o trecho básico que termina com "DO n ...."

B - é a parte do grafo correspondente aos demais comandos da sequência

C - é o vértice que representa o comando com rótulo n (que, por definição da linguagem, não pode ser um comando de transferência).

Cada módulo de execução (subrotina, função ou programa principal) é dado como entrada para um programa que o "resume" numa tabela. A partir dessa tabela é construído o grafo correspondente a esse módulo.

Para cada trecho básico do programa, são colocadas na tabela as seguintes informações:

a- rótulo inicial do trecho básico (se houver), a partir do qual serão determinados os arcos que "chegam" ao vértice correspondente

b- tipo do último comando do trecho básico

c- lista de rótulos para os quais o último comando pode desviar (portanto, a partir de b- e c- podemos determinar os arcos que partem de cada vértice)

d- localização da linha final do trecho - usada na eventual inserção de um contador nesse trecho

Para construção dessa tabela, o programa é lido por um analisador sintático que retorna, para cada comando

a- rótulo do comando (se houver)

b- tipo do comando

c- lista de rótulos para os quais o comando pode eventualmente desviar

c- informações adicionais como nome da subrotina no caso do comando "CALL", nome da variável à qual é atribuído um rótulo, no caso do comando "GO TO variável", etc.

O algoritmo A1, abaixo, é usado para a construção da tabela de trechos básicos.

```
repita
   obtenha comando;
   se comando lido tem rótulo então execute Q1;
   caso comando lido seja
      IF, GO TO {ou qualquer outro comando que envolve
      desvios}   :  execute Q2;
      DO         :  execute Q3;
      outros     :  ;
      fim  caso comando lido ...
até que comando lido seja  END
```

algoritmo A1

Observação:

Neste trabalho, os algoritmos são apresentados numa linguagem onde os passos elementares são descritos por sentenças em português e o controle e sequência de execução são descritos por construções semelhantes às encontradas em linguagens da família do ALGOL. As construções deste "ALGOL-informal", que dispensam maiores informações são:

se <condição> então <comando$_1$> senão <comando$_2$>

repita <sequência de comandos> até <condição>

caso <variável> seja

<expressão$_1$> : <comando$_1$>;

<expressão$_2$> : <comando$_2$>;

⋮ ⋮

<expressão$_n$> : <comando$_n$>

fim

início <sequência de comandos> fim

Os algoritmos Q1, Q2, Q3 e Q4 (utilizado em Q1) são descritos a seguir:

início {foi encontrado um comando rotulado que pode demarcar o início de um novo trecho básico}

se comando lido ≠ FORMAT então

início

execute Q4;

se o comando anterior nâo envolver desvios

então marque na tabela o início de um novo trecho básico {se o comando anterior envolver desvios esta marcação já foi feita por Q2};

coloque na tabela o rótulo do comando lido como sendo o rótulo inicial do trecho

fim

fim {de Q1}

algoritmo Q1

início { o comando lido envolve desvios, portanto termina um trecho básico }

guarde na tabela a lista de rótulos para os quais o comando pode desviar;

marque na tabela o início de um novo trecho básico

fim {de Q2}

algoritmo Q2

início {o comando lido é DO}

marque na tabela o início de um novo trecho básico;

empilhe o rótulo do comando que encerra a "malha" correspondente e o índice do trecho atual, numa pilha auxiliar

fim {de Q3}

algoritmo Q3

início { verificar se o rótulo do comando lido termina a "malha" de algum comando DO }

fecha := falso;

enquanto o rótulo do comando lido for igual ao rótulo no topo da pilha

faça início

fecha := verdade;

guarde o índice do trecho atual e o índice no topo da pilha numa tabela auxiliar ;

desempilhe o rótulo e o índice

fim;

se fecha {o comando lido fecha alguma malha}

então marque na tabela o início de um novo trecho

fim de Q4

algorimo Q4

DEFINIÇÃO DOS ARCOS

Tendo construído a tabela de blocos básicos, os arcos do grafo podem ser definidos através do seguinte algoritmo:

início

para todo trecho B na tabela faça

caso último comando de B seja

DO, CONTINUE, comando aritmético, CALL,

{ou qualquer outro que não envolva desvios} : defina um arco de B para o bloco seguinte;

IF, GOTO {comandos que sempre envolvem desvios} :

início

para todo rótulo R para os quais o último comando de B pode desviar faça

início

k:= trecho cujo rótulo inicial é R;

defina um arco de B para k

fim;

algoritmo A2

```
        if lógico: início
                   defina dois arcos, saindo de B para os dois
                   trechos seguintes a êle na tabela
                   fim ;
        fim {caso último comando... } ;
        para todo par de blocos (B1,B2) na tabela auxiliar, monta
                da por Q4 faça  defina um arco de B2 para B1
                                {esses arcos "fecham" os ciclos
                                 correspondentes às "malhas" dos
                                 comandos DO do programa}
fim  {de A2}
```

algoritmo A2 - continuação

2.2 Inserção dos Contadores no Programa

A inserção dos contadores no programa deve ser feita de forma que as modificações introduzidas não alterem o resultado final do mesmo.

Podem-se distinguir tres tipos distintos de trechos básicos, estabelecendo-se para cada tipo uma substituição que permita inserir os medidores de fluxo sem introduzir outras alterações no comportamento original do programa:

I - Trecho básico em que o último comando não é if-lógico nem DO:

Neste caso, o trecho básico é da forma:

<rótulo> <comando$_1$>
<comando$_2$>
⋮
<comando$_n$>

(rótulo pode ser omitido)

e é substituído por

<rótulo> CONT(i) = CONT(i) + 1
< comando$_1$ >
< comando$_2$ >
⋮
< comando$_n$ >

onde CONT(i) é o contador de frequência.

II - Trecho básico resultante de if-lógico.

O comando if-lógico é da forma:

<rótulo> IF(<expressão lógica>) <comando>

Como já foi dito anteriormente, <comando> é tratado como sendo um trecho básico independente e pode ter sua frequência medida. Uma forma de realizar tal medição é substituir todo o comando por

rótulo IF(< expressão lógica>) CONT(i)=CONT(i)+1
IF(< expressão lógica>) <comando>

Essa substituição apresenta as seguintes desvantagens:

1 - Ineficiência : o cálculo de expressão lógica é repetido desnecessáriamente.

2 - Efeito lateral: <expressão lógica> pode usar funções que alteram o valor de alguma variável do programa. Portanto o cálculo repetido da mesma pode alterar o resultado do programa original.

A substituição adotada neste caso, mais segura e eficiente foi:

rótulo VAR = <expressão lógica>

IF(VAR) CONT(i) = CONT(i) + 1

IF(VAR)<comando>

(VAR é uma variável que não aparece no programa original).

III - Trecho básico resultante de comando DO:

O comando DO, da forma

DO <rótulo> <variável> = <expressão$_1$>, <expressão$_2$>

⋮

<rótulo> <comando$_1$>

é substituído por

DO <rótulo'><variável> =<expressão$_1$>,<expressão$_2$>

⋮

<rótulo> <comando$_1$>

<rótulo'> CONT(i) = CONT(i) + 1

onde <rótulo'> é um número de comando que não aparece em nenhum outro ponto do programa.

CAPITULO 3

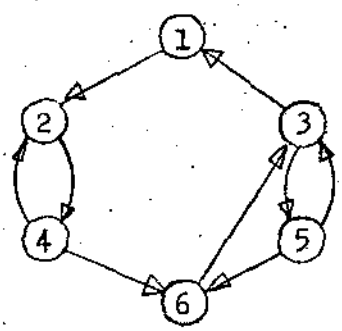
Minimização do número de pontos de medição de fluxo num grafo

No capítulo anterior, mostramos que para determinação do número de vezes que cada trecho de um programa é executado, esse programa será representado por um grafo dirigido, onde os vértices representam os trechos básicos e as arestas representam os desvios entre eles.

Considerando o número de vezes que cada trecho básico é executado como sendo o fluxo através do vértice correspondente, é evidente que todos os vértices, exceto os vértices inicial e final satisfazem o princípio da conservação de fluxo: o total de fluxo que "chega" a cada um dos vértices é igual ao total de fluxo que "sai" desse mesmo vértice. Como o total de fluxo que sai do vértice inicial é igual ao total de fluxo que chega ao vértice final, podemos adicionar um arco conectando os dois, de forma a obter um grafo de fluxo conservativo.

A partir desse ponto, os métodos de teoria dos grafos podem ser usados para determinar um conjunto mínimo de pontos de medição de fluxo, bem como as equações que permitirão calcular os fluxos que não são medidos em função dos demais. O método aqui exposto foi proposto por Knuth[4], sem uma formalização rigorosa. Neste capítulo é apresentado um tratamento formal do problema bem como a prova de que o método leva à minimização do número de pontos de medição

Considere-se o grafo G1 da figura abaixo



Supondo que seja um grafo de fluxo conservativo, podemos aplicar a primeira lei de Kirchoff: o total de fluxo que chega a cada vértice é igual ao total de fluxo que sai desse mesmo vértice, obtendo equações como:

$f_{1,2} = f_{3,1}$ ($f_{1,j}$ representa o fluxo na aresta que liga o vértice i ao vértice j)

$f_{3,1} + f_{3,5} = f_{6,3} + f_{5,3}$

etc...

A rigor, não estamos interessados em conhecer o valor dessas incógnitas mas sim a sua soma em cada um dos lados dessas equações (ou seja, estamos interessados no fluxo em cada vértice). O problema da obtenção dos fluxos nos vértices de um grafo pode ser reduzido ao problema da obtenção dos fluxos nas arestas de outro grafo, como é mostrado a seguir.

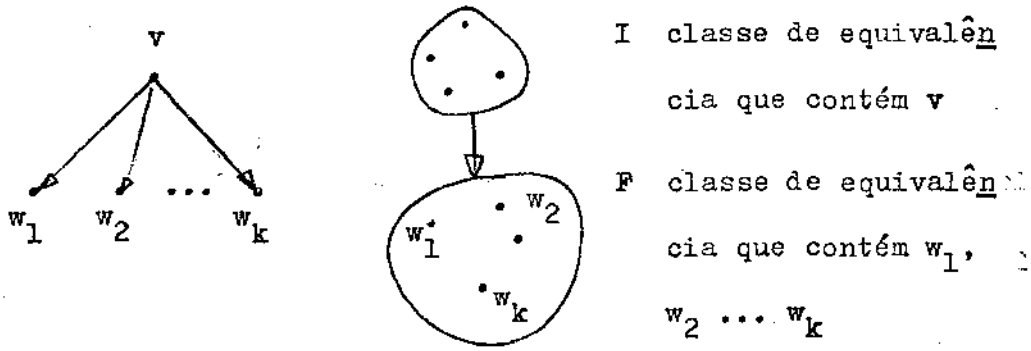
Dado um grafo G = (V, E), onde V é o conjunto de vértices e E é o conjunto de arestas, constrói-se o grafo reduzido da seguinte maneira:

a - define-se uma relação R_o sobre V como sendo: para y_1 e $y_2 \in V$, $y_1 \ R_o \ y_2$ se e somente se existem arestas (x,y_1) e (x,y_2), para algum $x \in V$, ou se $y_1 = y_2$.

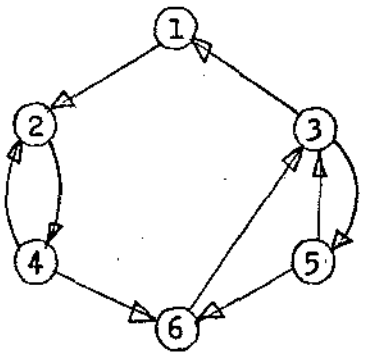
b - define-se a relação R como o fecho transitivo de R_o. Como R_o é reflexiva e simétrica, R é uma relação de equivalência.

c - define-se V' como o conjunto das classes de equivalência induzidas por R sobre V.

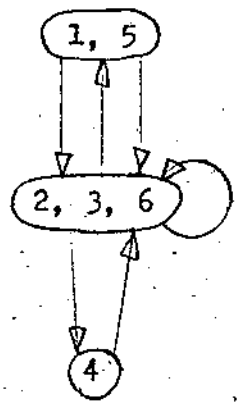
d - constrói-se E' com n arestas, onde n é o número de vértices de V e estabelece-se uma correspondencia entre as arestas de E' e os vértices de V: dado um vértice v de V, a aresta A_v, correspondente em E' tem como extremos inicial e final os vértices I e F de V', respectivamente, onde I é a classe de equivalência que contém v e F é a classe de equivalência que contém os extremos finais das arestas de E que tem v como extremo inicial (tais arestas pertencem à mesma classe de equivalência, pela definição de R). A figura abaixo ilustra a construção de uma aresta de E'.



Exemplo de redução



a- grafo original



b- grafo reduzido

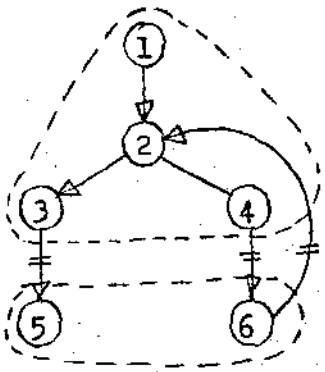
Por construção, o fluxo numa aresta A_v, no grafo reduzido corresponde à soma dos fluxos que saem de v, no grafo original. Com isso, o problema de "fluxo em vértices" foi reduzido a um problema de "fluxo em arestas".

Definição 3.1 - Seja G = (V , E) um grafo conexo, onde V é o conjunto de vértices e E é o conjunto de arestas. Dizemos que C, subconjunto de E, é um corte em G se existe W, subconjunto de V tal que

a - todas as arestas de C são da forma (x_1,x_2) ou (x_2,x_1) onde $x_1 \in W$ e $x_2 \in V - W$.

b - todas as arestas de E - C são da forma (x_1,x_2) onde x_1 e $x_2 \in W$ ou x_1 e $x_2 \in V - W$.

Exemplo:



as arestas marcadas com '=' constituem um corte.

Propriedade 3.1 - Seja G um grafo dirigido, C um corte em G e (W, V-W) a partição em V decorrente de C. Se G é de fluxo conservativo, é fácil ver que a soma dos fluxos nas arestas de C que chegam a W é igual à soma dos fluxos nas arestas de C que partem de W.

Definição 3.2 - Seja G = (V,E) um grafo conexo e T = (V,E'), com $E' \subseteq E$, um subgrafo de G. O subgrafo T é uma árvore geradora de G se

a - T não contém ciclos

b - T é um grafo conexo

Algoritmo 3.2 (algoritmo de Kruskal para construir uma árvore geradora de um grafo G)

Entrada : o grafo G = (V,E)

Saída : E' E, conjunto das arestas que constituem uma árvore geradora de G

início

$E' := \{\}$;

para toda aresta a em E faça

se a não forma ciclo com as arestas de E' então $E' := E' \cup \{a\}$

fim;

Lema 3.2 - Seja G = (V,E) um grafo dirigido, de fluxo conservativo e T = (V,E') uma árvore geradora de G. Para toda aresta $a \in E'$, o fluxo em a é igual à soma algébrica dos fluxos em todas as arestas b de E-E' tais que $T + \{b\}$ tem um ciclo passando por a.

Prova : Como T é uma árvore geradora, qualquer aresta $a \in E'$ constitue um corte em T. Seja (W, V-W) a partição em V decorrente desse corte.

Uma aresta $b \in E - E'$ tem um vértice em W e outro em V-W se e somente se b forma um ciclo com arestas em E' e passando por a.

Portando $C = \{a\} \cup \{b \mid b \in E-E' \text{ e } T+\{b\} \text{ tem um ciclo passando por a}\}$ constitue um corte em G.

Assim sendo, pela propriedade 3.1, a soma algébrica dos fluxos nas arestas de um corte é zero, completando a prova.

Com base no lema acima e no algoritmo de Kruskal, construiu-se o algoritmo abaixo, que permite determinar, além de uma árvore geradora, as equações que dão os fluxos nas arestas dessa

árvore geradora em função dos fluxos nas demais arestas.

Algoritmo 3.3

Entrada: um grafo G = (V, E)

Saída: uma árvore geradora de G e o conjunto de equações que permitem calcular os fluxos nas arestas da árvore em função dos fluxos nas demais arestas.

início

E' := { } ;

para toda aresta a ∈ E faça

se a não forma ciclo com as arestas em E'

então E' := E' ∪ {a}

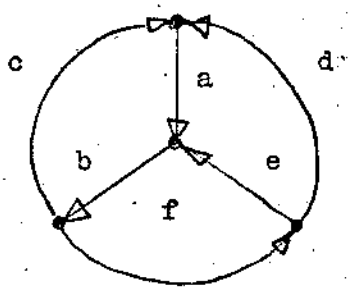
senão para toda aresta b no ciclo passando por a

faça acrescente '± F(a)' na equação para calcular F(b). Se, no ciclo, as arestas a e b tiverem o mesmo sentido (ambas horárias ou ambas anti-horárias), o sinal usado será '+' e '-' caso contrário.

Assim sendo, para obter o valor de todos os fluxos em um grafo não é necessário medir cada fluxo : uma vez construída uma árvore geradora através da aplicação do algoritmo 3.3, bastará medir o fluxo em todas as arestas não pertencentes a essa árvore; os demais fluxos serão calculados a partir das equações montadas pelo algoritmo.

Exemplo de uso do Algoritmo 3.3

Considere-se o grafo da figura abaixo, supondo que a ordem de entrada das arestas para o algoritmo seja a,b,c,d,e,f.



após a execução de cada passo teremos:

passo	aresta	pertence à árvore geradora?	conj. E'	sistema de equações
1	a	sim	a	
2	b	sim	a,b	
3	c	não	a,b	$\begin{cases} F(a)=+F(c) \\ F(b)=+F(c) \end{cases}$
4	d	sim	a,b,d	$\begin{cases} F(a)=+F(c) \\ F(b)=+F(c) \end{cases}$
5	e	não	a,b,d	$\begin{cases} F(a)=+F(c)-F(e) \\ F(b)=+F(c) \\ F(d)=-F(e) \end{cases}$
6	f	não	a,b,d	$\begin{cases} F(a)=+F(c)-F(e)+F(f) \\ F(b)=+F(c)+F(f) \\ F(d)=-F(e)+F(f) \end{cases}$

Portanto, se medirmos F(c), F(e) e F(f), os demais fluxos podem ser calculados pelas equações acima.

Propriedade 3.4 O número de medições requeridas pelo método apresentado, ($|E| - |V| + 1$) é mínimo.

Prova

a - Seja $G = (V, E)$ um grafo conexo. Se $|E| > |V| - 1$ então G tem pelo menos um ciclo ($|E|$ representa a cardinalidade do conjunto E).

b - Seja $G = (V, E)$ um grafo conexo e de fluxo conservativo e suponhamos que seja possível determinar todos os f_a, onde f_a representa o fluxo na aresta a, utilizando-se $M < |E| - |V| + 1$ medições.

Neste caso, não foram medidos os fluxos em $N > |V| - 1$ arestas, já que $M + N = |E|$. Sendo G um grafo conexo, o conjunto de $N > |V| - 1$ arestas cujos fluxos não foram medidos, inclui pelo menos um ciclo. Mostremos que, no caso geral, os fluxos nesse ciclo não podem ser univocamente determinados, contrariando a hipótese:

Em nosso caso, os fluxos correspondem ao número de execuções de determinado trecho de programa, logo podem ser positivos ou nulos. Como não se dispõe "a priori" de informação adicional sobre os valores dos fluxos, o método em questão deve funcionar, em particular, para o caso em que todos os fluxos sejam positivos.

Percorra-se o ciclo mencionado, alterando cada fluxo de acordo com a seguinte regra:

$f'_a = f_a + k$ se a orientação da aresta concorda com o sentido de percurso do ciclo

$f'_a = f_a - k$ se não concorda

onde $\min f_a \geq k > 0$, o que é possível pois todos os f_a são positivos.

Estas alterações mantém a conservação de fluxo em cada vértice, uma vez que a soma algébrica das alterações em cada vértice do ciclo é zero. Assim sendo k corresponde a uma indeterminação nos valores do fluxo considerado.

Fica assim provada a impossibilidade de determinar univocamente os fluxos em todas as arestas de um ciclo, sem medir pelo menos o fluxo numa das arestas.

CAPÍTULO 4

Detalhes de Implementação de Algumas Partes do Sistema

Este capítulo mostra, em linhas gerais, alguns aspectos da implementação do sistema, considerados importantes.

4.1 Reconhecimento de comandos FORTRAN

O reconhecimento de comandos FORTRAN não pode ser feito através do esquema usual, onde se faz uma separação bem nítida entre análise léxica e sintática porque:

- caracteres brancos não são separadores de unidades léxicas.
- não existem palavras reservadas; dependendo do contexto, "DO", "IF", "READ", "DATA", etc. podem ser palavras-chave em comandos ou parte de um identificador.

Ainda assim, na construção do reconhecedor, procurou-se separar o máximo possível as funções léxica e sintática.

O reconhecedor construído consiste basicamente de:

a- um "analisador léxico" que lê o programa FORTRAN, ignorando espaços em branco, cartões de continuação, etc.. Reconhece as seguintes categorias de símbolos:

   - cadeias de caracteres alfabéticos
   - cadeias de caracteres numéricos
   - delimitadores

b- um "analisador sintático" - devido às peculiaridades do FORTRAN os métodos clássicos de análise sintática não são eficientes no

seu reconhecimento.

O analisador léxico "devolve" ao analisador sintático um comando FORTRAN "dissecado" nas seguintes partes:

rótulo cadeia de carac. alfabéticos outros símbolos

O analisador tenta, em primeiro lugar, reconhecer em cadèia de caracteres alfabéticos alguma plavra-chave. Se os caracteres iniciais da cadeia não formarem uma palavra-chave, o analisador decide que está diante de um comando aritmético e tenta reconhece-lo como tal. Em caso contrário, a palavra-chave encontrada indica o tipo de comando que deve ser reconhecido.

Em alguns casos, o analisador deve fazer retrocesso. Por exemplo, nas sequências de caracteres abaixo, as palavras chave não suficientes para identificar o tipo de comando:

```
DO 10 I = 1 ...
IF ( J + 1 ) ...
DATA ALFA ...
COMMON C(10) ...
READ ( 2, 1 ) ...
```

Nestes casos, o analisador supõe inicialmente que o comando é do tipo indicado pela palavra-chave e se no decorrer da análise aparecer algum erro, recomeça a análise, supondo que está diante de um comando aritmético.

Este esquema não traz maiores problemas uma vez que se supõe que os programas de entrada são sintaticamente corretos.

4.2 Implementação dos Algoritmos que Manipulam Grafos

A localização dos pontos de medição de fluxo num grafo, descrita no capítulo 3, é feita em duas etapas: redução do grafo e construção da árvore geradora.

4.2.1 Redução do Grafo

Conforme o exposto no capítulo precedente, a redução do grafo é feita construindo-se as classes de equivalência de vértices e as arestas representativas de cada vértice. As classes de equivalência são conjuntos sobre os quais serão efetuadas operações do tipo: "encontrar a classe a que pertence um vértice", "unir duas classes de equivalência", etc..

As classes de equivalência serão representadas por árvores, onde os nós são os vértices e a raiz é o vértice considerado representativo da classe.

O grafo reduzido será uma coleção de vértices, onde cada vértice v terá os seguintes atributos:

pai(v) - é o vértice antecessor a v na estrutura que representa a classe de equivalência à qual v pertence.

filho(v) - é o vértice que designa a aresta representativa do vértice v. Esta aresta liga v a filho(v).

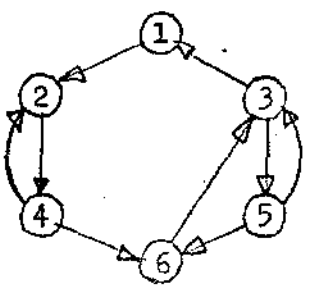
A função super(v) calcula o vértice representativo da classe de equivalência a que pertence um vértice v:

```
função super (v : vértice);
se  pai ( v ) = nenhum  então  super := v
                       senão super := super(pai(v));
```

Para cada aresta (v,w), no grafo original, é chamado o procedimento pegaaresta(v,w), abaixo, que monta, a partir das arestas, o grafo reduzido.

```
procedimento pegaaresta( v, w : vértices );
se filho(v) = nenhum então filho(v) := w
   senão {faça w e filho(v) equivalentes}
    início
     w := super(w);
     se w ≠ super(filho(v))
         então  pai(w) := super(filho(v))
fim;
```

Exemplo:



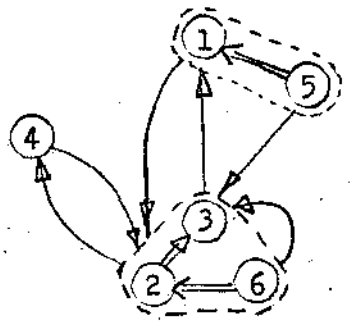
grafo original

(1,2)
(2,4)
(3,1)
(3,5)
(4,2)
(4,6)
(5,3)
(5,6)
(6,3)

ordem de entrada dos arcos

vértice	pai	filho
1	nenhum	2
2	3	4
3	nenhum	1
4	nenhum	2
5	1	3
6	2	3

atributos "pai" e "filho" de cada vértice após a entrada dos arcos



grafo reduzido - as setas duplas representam o atributo "pai" e as setas simples representam o atributo "filho"

4.2.2 Construção da Árvore Geradora e Montagem das Equações de Fluxo

A construção da árvore geradora (do grafo reduzido), é feita usando o algoritmo de Kruskal modificado (algoritmo 3.3). Esse algoritmo tenta colocar cada aresta num conjunto de arestas que ao final constituirão uma árvore geradora. No nosso caso, as arestas serão representadas pelos vértices do grafo original. Os vértices do grafo reduzido são as classes de equivalência dos vértices originais e cada classe é representada por um vértice dito representativo dessa classe, ao qual chamaremos de "supervértice".

Cada um desses supervértices terá os seguintes atributos adicionais:

superpai - indica o antecessor do vértice na estrutura que representa a árvore geradora.

u - a classe representada por um supervértice v será a extremidade inicial de uma aresta da árvore geradora. O atributo u(v) é usado para designar as extremidades dessa aresta, que são super(u(v)) e super(filho(u(v))).

d - é usado para indicar se o sentido do fluxo na aresta designada por u(v) coincide ou não com a orientação dada à aresta.

Além desses atributos, "medir" é usado para todas as arestas (vértices no grafo original) para indicar se o fluxo correspondente deve ser medido ou não.

O núcleo do programa para a construção do grafo será:

```
para todo vértice v do grafo original
        faça inserir ( v ) ;
```

O procedimento inserir (v) tenta colocar cada vértice do grafo original, que representa uma aresta do grafo reduzido, na árvore geradora:

```
procedimento inserir ( v : vértice); v1, w : vértices;
início  v1 := super(v);
        mudaraiz(v1); {v1 passa a ser raiz da subárvore a que
                       pertence}
        w := super (filho(v)); {v1 e w são as extremidades da
                                aresta correspondente a v}
  se v1 ≠ raiz(w) {v não forma ciclo com as arestas das subár-
                   vores}
     então  {juntar as subárvores a que pertencem v e w}
       início
       superpai(v1) := w;
       d(v1) := '+';
       u(v1) := v;
       medir(v) := 'não';
       fim
     senão  {atualize as equações para calcular os fluxos nas
             arestas da árvore}
     início  medir(v) := 'sim';
      enquanto w ≠ v1 faça
         início
         acrescente  d(w) e f(v) na equação para calcular o flu
         xo de  u(w);
         w := superpai(w)
         fim
   fim
fim {de inserir};
```

O procedimento mudaraiz (v) transforma a subárvore que contem v, de forma que o vértice v passe a ser a raiz da mesma:

```
procedimento mudaraiz( v : vértice);
se superpai(v) ≠ nenhum  { v não é raiz da subárvore }
  então  início
         mudaraiz(superpai(v));
         superpai(superpai(v)) := v;
         d(superpai(v)) := se d(v) = '+' então '-'
                                          senão '+';
         u(superpai(v)) := u(v);
         superpai(v) := nenhum    { v passa a ser raiz }
         fim;
```

A função raiz é usada de modo análogo a super, na reduzão do grafo e acha a raiz da subárvore à qual pertence a aresta correspondente ao vértice v:

```
função raiz ( v : vértice) : vértice;
se superpai(v) = nenhum então  raiz := v
                       senão  raiz := raiz(superpai(v));
```

CAPÍTULO 5

Estimativa do Tempo de Execução de Comandos

Os capítulos anteriores foram orientados no sentido de mostrar uma forma eficiente de medir frequências de execução de comandos. Uma informação mais útil ao programador seria o tempo total gasto na execução de cada comando.

Este capítulo descreve como pode ser feita a estimativa do tempo de execução dos vários comandos FORTRAN, através de uma análise simplificada.

Para avaliar com precisão o tempo de execução de cada comando de um programa, o ideal seria conhecer exatamente o código gerado pelo compilador.

Essa não é uma tarefa simples, principalmente quando se trata de um compilador com otimização de código. A construção de um sistema nesses padrões seria tão complicada quando a construção do próprio compilador, o que reforça a afirmação feita por Knuth[4], de que a avaliação do tempo gasto na execução dos comandos deveria ser uma opção nos compiladores normais.

Como o nosso objetivo é estimar "por alto" o tempo de execução, optamos por uma solução bem mais simples: a cada tipo de comando associamos um peso constante, e para cada operador, parâmetro, etc., associamos um peso que é acrescido ao peso total do comando.

Observações

a - Como o computador usado tem circuitos especiais para as operações com valores reais, o tempo de execução das instruçoes aritméticas com valores reais não difere muito das correspondentes operações com inteiros e a tarefa de estimar o tempo gasto no cálculo de expressões aritméticas foi bastante facilitada.

b - numa expressao aritmética podem ser feitas chamadas a funçoes que podem ser funções de biblioteca do FORTRAN ou funções escritas pelo usuário. No caso das funções de biblioteca, o tempo estimado, determinado experimentalmente, é computado no tempo da expressão. No caso das funções escritas pelo usuário, somente é computado o tempo correspondente à chamada; qualquer função escrita pelo usuário, porém, pode fazer parte do conjunto de programas submetidos ao sistema, tendo assim avaliados os tempos gastos na execução de cada um de seus comandos.

c - variáveis indexadas podem ocorrer num comando e seus índices podem ser expressões. Nestes casos, o tempo de cálculo dessas expressoes também é computado no peso total do comando, mesmo que, por exemplo, a expressão ocorra à esquerda do sinal "=" num comando aritmético.

d - variáves em dupla precisão e complexos não foram considerados nesta implementação e portanto eventuais operações envolvendo tais variáveis tem seu tempo estimado como se tratasse de variáveis simples.

A determinação desses pesos foi feita experimentalmente e os testes realizados mostraram que, na maioria dos casos, a margem de erro é menos que 30%, o que pode ser considerado muito bom, se levarmos em conta a simplicidade do método.

A seguir mostramos como é feita a estimativa do tempo de execução de alguns comandos FORTRAN.

5.1 Comando Aritmético

O comando aritmético, da forma

< variável > = < expressão >

tem seu tempo estimado dado por $T_{arit} + T_{expr}$, onde

T_{arit} - é o peso constante associado ao comando aritmético. Esse peso foi obtido experimentalmente medindo-se o tempo de execução do comando aritmético na sua forma mais simples, por exemplo A = B .

T_{expr} - é o tempo estimado para o cálculo da expressão à direita do sinal "=", obtido somando-se os pesos associados a cada uma das operações envolvidas, sem levar em conta a ordem em que são efetuadas ou mesmo eventuais operações com resultados intermediários. Por exemplo, o tempo estimado para o cálculo da expressão A + B * 3.5 seria dado por $T_{expr} = T_{adição} + T_{mult}$

5.2 Comando DO

No caso do comando DO, é estimado apenas o tempo relativo à inicialização do comando, sendo que o tempo de incremento e teste da variável contadora é atribuído ao comando que encerra a "malha" correspondente.

O tempo estimado para o comando DO, da forma

DO <rótulo> <var> = <$expr_1$> , <$expr_2$> , <$expr_3$>

é dado por $T_{do} + T_{expr_1} + T_{expr_2} + T_{expr_3}$

5.3 Comando "IF-lógico"

O comando "IF-lógico", da forma

IF (<expressão lógica>) <comando>

tem o seu tempo estimado dado por $T_{if} + T_{expr}$. O tempo estimado de <comando> é calculado independentemente.

5.4 Comando CALL

O tempo estimado do comando CALL depende da subrotina chamada:

- se a subrotina faz parte da biblioteca do FORTRAN, o tempo correspondente à execução da mesma é computado.
- se a subrotina chamada é uma rotina du usuário, somente o tempo de transferência para a subrotina é considerado.

O tempo de execução do comando CALL ainda depende do nú

mero de parâmetros da subrotina. A cada parâmetro é associado um peso constante, determinado experimentalmente, que é acrescido ao peso total do comando. Além disso, na lista de parâmetros podem ocorrer expressões cujo tempo estimado também é acrescido ao peso total do comando.

5.5 - Comandos de Entrada e Saída

Como o tempo de execução dos comandos de entrada e saída depende de vários fatores externos ao programa e de fatores que só podem ser determinados dinamicamente, como repetições implícitas, especificações de formato dinâmicas, etc., não é possível estimar, através de uma análise estática o seu tempo de execução. Optou-se portanto por não estimar esses valores. A solução alternativa seria medi-los dinâmicamente, consultando o relógio da máquina.

Os comandos ENCODE e DECODE, por motivo semelhante, também não tem o seu tempo de execução estimado.

Observação:

os comandos acima podem encerrar a "malha" de algum comando DO. Neste caso, o tempo de teste e incremento da variável contadora é considerado como o peso total do comando.

CAPÍTULO 6

Resultados e Conclusões

6.1 O Sistema Implementado

O sistema de avaliação descrito foi implementado no computador DEC-10 da UNICAMP e consta, em linhas gerais, dos seguintes passos:

- Ler o programa FORTRAN a ser avaliado.
- Gerar um a versão modificada do programa, equivalente a êle, a menosdo fato de realizar a avaliação.
- Gerar um arquivo auxiliar, contendo informações auxiliares como tempo estimado de execução de cada comando, relação das frequências que não precisam ser medidas e equações para cálculo das mesmas.

O programa modificado é então executado e a seu término é automáticamente ativada uma rotina que emite um relatório onde ao lado de cada comando do programa original aparecem o tempo estimado total gasto na sua execução e a frequência correspondente.

6.2 Uso do Sistema

Na descrição sumária abaixo da utilização do sistema implementado na UNICAMP, supos-se uma familiarizaçãc do leitor com os principais comandos do sistema operacional do DEC-10 [9].

Para execução por terminal, o comando

```
.R AVAL
```

faz com que o sistema seja executado, pedindo ao usuário que dê <nome>.<ext> do arquivo que contém o programa a ser avaliado. Este arquivo, além do programa principal, deve conter o conjunto de todas as rotinas e funções que se deseja avaliar.

Ao gerar a versão modificada do programa, o sistema introduz nomes e rótulos no programa; os nomes introduzidos se iniciam sempre com o prefixo "IPTW" e os rótulos gerados sempre são maiores que 99000. Para evitar conflitos é conveniente que o usuário não use nomes e rótulos que possam coincidir com aqueles eventualmente introduzidos no programa.

O programa modificado pode ser executado através do comando

.EXECUTE <nome>.PRG

Durante a execução do programa modificado, deve estar disponível o arquivo auxiliar (<nome>.DAT) criado por AVAL.

O programa modificado, ao ser executado, cria o arquivo <nome>.LST, que contém o relatório da avaliação.

O apêndice 2 contém exemplos comentados da aplicação do sistema na avaliação de execução de alguns programas típicos de FORTRAN.

6.3 Conclusões

A avaliação dos tempos de execução só poderia ser feita com precisão se fosse conhecido o código gerado pelo compilador a partir de cada comando. A avaliação oferecida pelo sistema, embora apresente valores coerentes com os valores reais, deve portanto ser considerada grosseira (dentro de 30% tipicamente) e encarada pelo usuário com a devida cautela. A frequência de execução dos comandos entretanto, é absolutamente precisa.

A localização de um conjunto mínimo de pontos de medição em programas FORTRAN e a inserção dos contadores deve ser feita em dois passos, pois todo o grafo deve ser conhecido a fim de se determinar o conjunto mínimo de pontos de medição. Este fato ocorre devido à falta de uma estrutura melhor definida nos comandos FORTRAN.

Se os comandos da linguagem obedecem a uma estrutura mais rígida (por exemplo Algol ou Pascal, sem o uso de "GO TO") é possível determinar os pontos de medição e inserir os contadores num único passo. Um sistema baseado nessa propriedade é descrito em [8], onde é apresentado um sistema de avaliação para SIMULA.

6.4 Sujestoes para Futura Pesquisa

Uma tendência recente no projeto de linguagens de programação é o uso de máquinas hipotéticas para definir tanto a semântica da linguagem como para implementar as versões-piloto dos compiladores, a partir das quais serão feitas as implementações em máquinas reais (ex. a máquina P do Pascal).

Parece pois ser promissor o desenvolvimento de ferramentas como a descrita neste trabalho, baseadas não numa máquina real, e sim na máquina hipotética orientada para a linguagem. Esta abordagem estaria em concordância com outra tendencia atual que é a ênfase em portabilidade em sistemas de apoio à programação.

No caso da avaliação dos tempos de execução de comandos, o único parâmetro dependente da implementação seria o tempo de execução de cada instrução da máquina hipotética, que poderia ser facilmente determinado pelo implementador.

APENDICE 1

Pesos Associados aos Comandos e Operadores em FORTRAN

Os valores abaixo foram obtidos experimentalmente através da execução repetida dos vários comandos nas suas várias formas e consultando o relógio da máquina

A1.1 Pesos Associados a Comandos

Comando	peso (s)
GOTO	1.7
IF arit.	2.7
IF lógico	4.0
CALL	18.5
Aritmético (A=B)	5.0

A1.2 Pesos Associados a Operadores em Expressões

A1.2.1 Operadores Aritméticos

Operador	peso (s)
+ (soma)	3.0
- (subtração)	3.0
(multiplicação)	3.6
(exponenciação)	3.6 a 440.0

A1.2.2 Operadores Lógicos

Operador	peso (s)
GT	4.5
GE	4.5
LT	4.5
LE	4.0
EQ	3.0
NE	4.0
NOT	3.0
AND	2.5
OR	2.0
XOR	2.5

A1.2.3 Funções de Biblioteca

Função	peso (s)
ABS	30,0
ALOG	112.0
ALOG10	141.0
ASIN	240.0
ACOS	240.0
ATAN	52.0
COS	137.0
EXP	148.0
MOD	48.0
SQRT	77.0
SIN	147.0

APÊNDICE 2

Exemplos de Uso do Sistema

As listagens abaixo foram obtidas da avaliação de alguns programas. A primeira coluna corresponde ao tempo estimado de execução do comando e a segunda coluna corresponde à frequencia de execução.

```
                      C       RESOLUCAO DE UM SISTEMA DE EQUACOES LINEARES
                      C       PELO METODO DE ELIMINACAO DE GAUSS

                              COMMON A(50,50),B(50),X(50),N
                     1
                      C       ENTRADA DOS DADOS
                              READ(1,1)N
                      1       FORMAT(I)
                              DO 10 I=1,N
                              READ(1,2)(A(I,J),J=1,N)
                              READ(1,2)(B(I),I=1,N)
                      2       FORMAT(10F)
                              CALL ESCREVE
                              CALL TRIANG
                              WRITE(3,100)
                      100     FORMAT(1X,/,1X,'SISTEMA TRIANGULARIZADO',/)
                              CALL ESCREVE
                              CALL GAUSS
                              WRITE(3,101)(X(I),I=1,N)
                      101     FORMAT(1X,'VALORES DE X ', 10F9.5)
                              CALL EXIT
                              END


                              SUBROUTINE GAUSS
                              COMMON A(50,50),B(50),X(50),N
                              X(N)=B(N)/A(N,N)
                              DO 10 J=N-1,1,-1
                              SOMA = 0.0
                              DO 20 K=J+1,N
                      20      SOMA=SOMA + X(K)*A(J,K)
                      10      X(J) = (B(J)-SOMA)/A(J,J)
                              RETURN
                              END
```

```
                                      SUBROUTINE TRIANG
                                      COMMON A(50,50),B(50),X(50),N
      10.7          1                 DO 10 I=1,N-1
     245.0         49                 K = IPIVO(I)
     441.0         49                 IF ( K .NE. I)   CALL TROCA(I,K)
     567.5         25
     377.3         49                 DO 20 J=I+1,N
   42875.0       1225                 C = A(J,I)/A(I,I)
    9432.5       1225                 DO 30  K = I,N
 2923625.0      61250     30          A(J,K) = A(J,K) - C * A(I,K)
   38097.5       1225     20          B(J) = B(J) - C * B(I)
     220.5         49     10          CONTINUE
                    1                 RETURN
                                      END

                                      FUNCTION IPIVO(K)
                                      COMMON A(50,50),B(50),X(50),N
     245.0         49                 I=K
     686.0         49                 IF( K .EQ. N)  GO TO 11
       0.0          0
     377.3         49                 DO 10  J= K+1,N
   23275.0       1225                 IF(A(J,K) .GT. A(I,K) ) I=J
     125.0         25
    5512.5       1225     10          CONTINUE
     245.0         49     11          IPIVO = I
                   49                 RETURN
                                      END
```

```
                                SUBROUTINE TROCA (I,J)

                                COMMON A(50,50),B(50),X(50),N

     192.5          25          DO 10 K=1,N

   18750.0        1250          T=A(I,K)
   31250.0        1250          A(I,K)=A(J,K)
   18750.0        1250          A(J,K)=T

    5625.0        1250    10    CONTINUE

     250.0          25          T=B(I)
     375.0          25          B(I)=B(J)
     250.0          25          B(J)=T

                    25          RETURN

                                END



                                SUBROUTINE ESCREVE

                                COMMON A(50,50),B(50),X(50),N

                     2          WRITE(3,100)
                   100          FORMAT(1X, 'MATRIZ A E VETOR B ')

      15.4          2
     450.0        100          DO 10 I=1,N
                    10          WRITE(3,101)(A(I,J),J=1,N), B(I)

                   101          FORMAT(1X,11F9.5)

                     2          RETURN

                                END
```

```
                C       HEAPSORT

                        SUBROUTINE SIFT

                        COMMON X,L,R, A(100)
                        IMPLICIT INTEGER(A-Z)

  735.0     149         I=L
 1281.4     149         J=2*I
 1490.0     149         X=A(I)

 2392.0     598    12   IF( J.GT. R) GO TO 13
  154.7      91
 2028.0     507         IF( J.EQ. R) GO TO 11
    6.8       4
 7042.0     503         IF( A(J) .LT. A(J+1) )  J = J+1
 1656.0     207
 6084.0     507    11   IF(X .GE. A(J)) GO TO 13

   98.6      58
 6735.0     449         A(I) = A(J)
 2245.0     449         I=J
 3861.4     449         J=2*I
  763.3     449         GO TO 12

 1490.0     149    13   A(I) = X
            149         RETURN
                        END


                        COMMON X,L,R,A(100)
                        IMPLICIT INTEGER(A-Z)


               1        ACCEPT15,N
              15        FORMAT(15I)
               1        ACCEPT15,(A(I),I=1,N)

    5.0        1        R = N
   23.0        1        L = (N / 2) +1
  459.0       51    21  IF ( L .EQ. 1) GO TO 22
    1.7        1
  400.0       50        L = L-1
  925.0       50        CALL SIFT
   85.0       50        GO TO  21

               1    22   CONTINUE


  900.0      100    2   IF ( R .LE. 1 ) GO TO 1
    1.7        1
  990.0       99        X = A(1)
 1485.0       99        A(1) = A(R)
  990.0       99        A(R) = X
  792.0       99        R = R - 1
 1831.5       99        CALL SIFT
  168.3       99        GO TO 2

               1    1   TYPE20,(A(I),I=1,N)

              20        FORMAT(1X,10I5)
               1        STOP
                        END
```

```
                                      SUBROUTINE XINT ( RMA, DQ, TET, S )
     300.0        60                  PI = 3.141592654
     900.0        60                  RA = PI/4.0
     240.0        60                  IF ( TET.GE.RA ) GO TO 21
      76.5        45
      75.0        15                  N = 50
      25.5        15                  GO TO 22
     225.0        45            21    N = 100
   36132.0        60            22    F = ( DQ - (( RMA)/COS(-TET))**2)**1.5
     300.0        60                  S = F
     516.0        60                  P = 2.0*TET/N
     696.0        60                  XTET = -TET
     642.0        60                  DO 23 LK = 1,N-1
  138054.0      5190                  IS = LK - 2*(LK/2)
   69201.0      5190                  COEF = 2 + 2*IS
   25959.0      5190                  XTET = XTET + P
 3109848.0      5190                  F = ( DQ - ( (RMA)/COS(XTET) )**2)**1.5
   64875.0      5190            23    S = S + COEF*F
   35952.0        60                  F = ( DQ - ( (RMA)/COS(TET) )**2)**1.5
    1596.0        60                  S = ( S + F )*P/3.0
                  60                  RETURN
                                      END
                      C
                      C               XINT  -  INTEGRAL  DE F  DE  2*PI-TET  ATE'  TET
                      C
                                      SUBROUTINE XINT ( RMA, DQ, TET, S )
     325.0        65                  PI = 3.141592654
     975.0        65                  RA = PI/4.0
     260.0        65                  IF ( TET.GE.RA ) GO TO 8
     110.5        65
       0.0         0                  N = 200
       0.0         0                  GO TO 6
     559.0        65            8     RRA = RA*2.
     260.0        65                  IF ( TET.GE.RRA ) GO TO 9
     110.5        65
       0.0         0                  N = 150
       0.0         0                  GO TO 6
     559.0        65            9     RRA = RA*3.
     260.0        65                  IF ( TET.GE.RRA ) GO TO 7
      34.0        20
     225.0        45                  N = 100
      76.5        45                  GO TO 6
     100.0        20            7     N = 50
   38948.0        65            6     F = ( DQ - ((RMA)/COS( TET ))**2)**1.5
     325.0        65                  S = F
    1963.0        65                  P = ( PI*2.0 - 2.0*TET)/N
    1638.0        65                  XTET = TET
     695.5        65                  DO 10 LK = 1,N-1
  144571.0      5435                  IS = LK - 2*(LK/2)
   63046.0      5435                  COEF = 2 + 2*IS
   27175.0      5435                  XTET = XTET + P
 3256652.0      5435                  F = ( DQ - ((RMA)/COS( XTET ))**2)**1.5
   67937.5      5435            10    S = COEF*F + S
   39377.0        65                  F = ( DQ - ((RMA)/COS(PI*2.-TET))**2)**1.5
    1729.0        65                  S = ( S + F )*P/3.0
                  65                  RETURN
                                      END
```

```
     5.0         1          PI = 3.141592654
     7.7         1          DO  100  KK = 1,5
                 5          TYPE 10, KK
                10      FORMAT(///,3X,'ENTRE COM PARAMETROS DO TANQUE',I4
                        1,/,5X,' FORMATO  E')
                 5          ACCEPT 11, R, H, SL
                11      FORMAT ( 3E )
                       C
                       C        CALC.  DA  INT.
                       C
    93.0         5          P = 2.0*R/25.0
    25.0         5          A = 0.0
                 5          WRITE ( 33,13 ) KK, R, H, SL
                13      FORMAT(///81('*'),//6X,'TANQUE  N.',I2,5X,'R = ',E12.6,5X,'H
                        6 = ',E12.6,5X,'L = ',E12.6,//4X,'NIVEL (M)',6X,13X,'MASSA
                        7 (TON)'/14,50('-'))
    38.5         5          DO 18 I = 1,25
  1000.0       125          A = A + P
 53825.0       125          RSQ = R**2
 56775.0       125          D = ( RSQ + SL**2 )/( 2.0*SL )
  1000.0       125          RMA = R - A
  2500.0       125          X = ( RMA )/R
  4875.0       125          IF ( ABS(X).GT.(1.0E-5) ) GO TO 14
   212.5       125
     0.0         0          TET = PI/2.0
     0.0         0          GO TO 15
 71575.0       125       14 TET = ATAN( SQRT( 1. - X**2)/X)
 54950.0       125          IF ( TET.LE.0 ) GO TO 16
   110.5        65
 25836.0        60          DQ = D**2
  1260.0        60          HLD = H/2. + SL - D
  1410.0        60       15 CALL XINT ( RMA, DQ, TET, S )
  3588.0        60          T1 = TET*(2.0*(-(DQ-RSQ )**1.5)/3.0+(RSQ)*(HLD))
 46428.0        60          T2 = -( H/2.0 + SL - D )*(SIN(TET)/COS(TET))*((R - A )**2)
   900.0        60          T4 = S/3.0
  1392.0        60          V = 2.0*( T1 + T2 +  T4 )*1.8336
   102.0        60          GO TO 19
   325.0        65       16 TET = TET + PI
  1852.5        65          CALL XINT ( RMA, DQ, TET, S )
 31551.0        65          T1 = TET*(2.0*(D**3-(DQ-RSQ)**1.5)/3.0 + (RSQ)*(HLD))
 21307.0        65          T2 = -( HLD )*( SIN(TET)/COS(TET) )*( RMA**2 )
 29952.0        65          T3 = 2.0*(D**3)*(PI - TET)/3.0
  1170.0        65          T4 = -S/3.0
  1703.0        65          V = 2.0*( T1 + T2 + T3 + T4 )*1.8336
               125       19 WRITE ( 33,17 ) A, TET, V
                17      FORMAT(1X,E12.6,4X,E12.6,4X,E12.6)
   562.5       125       18 CONTINUE
    22.5         5      100 CONTINUE
    18.5         1          CALL EXIT
                            END
```

REFERÊNCIAS

[1] Armen Nahapetian, Node Flows in Graphs With Conservative Flow, Acta Informatica 3.1, 1973

[2] Daniel Ingalls, FETE - A FORTRAN Execution-Time Estimator, Stanford Computer Science Report, CS-204, 1971

[3] Daniel Ingalls, The Execution-Time Profile as a Programming Tool, in R. Rustin (ed.) Compiler Optimization, 2nd Courant Computer Science Symposium (1970), Prentice-Hall, 1972

[4] D. E. Knuth and F. R. Stevenson, Optimal Measurement Points for Program Frequency Counts, BIT 13, 1973

[5] D. E. Knuth, An Empirical Study of FORTRAN programs, Software Practice and Experience 1,1971

[6] D. E. Knuth, the Art of Computer Programming, Addison-Wesley, vol. 1, 1968

[7] J. Cohen and C. Zuckerman, Two Languages for Estimating Program Effiency, Comm. of ACM, jun. 1974

[8] S. Arnborg, A Note on the Assignment of Measurement Points for Frequency Counts in Structured Programs, Bit 14, 1974

[9] FORTRAN IV (F40) Programmer's Reference Manual, Digital Equipment Corporation, DEC-10-LFLMA-B-D