COLEÇÕES DE CONJUNTOS DISJUNTOS: OPERAÇÕES E ALGORITMOS

Oswaldo Vernet de Souza Pires

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

Prof. Lilian Markenzon, D.Sc.
(Presidente)

Prof. Jayme Luiz Szwarcfiter, Ph.D.

Prof. Paulo Roberto de Oliveira, D.Ing.

Prof. Nair Maria Maia de Abreu, D.Sc.

RIO DE JANEIRO, RJ - BRASIL
ABRIL DE 1990

PIRES, OSWALDO VERNET DE SOUZA

Coleções de Conjuntos Disjuntos:
Operações e Algoritmos. [Rio de Janeiro] 1990
vii, 110 p. 29,7 cm (COPPE/UFRJ, M.Sc.,
Engenharia de Sistemas, 1990)
Tese - Universidade Federal do Rio de
Janeiro, COPPE
1. Operações e algoritmos para manipular
coleções de conjuntos disjuntos.
I. COPPE/UFRJ II. Título (série).

AGRADECIMENTOS

Primeiramente, a minha orientadora Lilian Markenzon, pela longanimidade que me dispensou durante a orientação desta tese, tolerando minhas crises existenciais.

A Edna Cerbino de Moura, pelo valioso apoio e motivação que tem me dado.

A Luci Pirmez, pelas palavras de incentivo e os proveitosos conselhos.

A todos os que, de alguma forma, em qualquer plano, colaboraram para a consecução deste trabalho.

A todos os que, não podendo ou querendo colaborar, ao menos não atrapalharam.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para obtenção do grau de Mestre em Ciências (M.Sc.).

COLEÇÕES DE CONJUNTOS DISJUNTOS: OPERAÇÕES E ALGORITMOS

Oswaldo Vernet de Souza Pires

Abril de 1990

Orientadora: LILIAN MARKENZON

Programa: ENGENHARIA DE SISTEMAS E COMPUTAÇÃO

O trabalho aborda, de início, uma breve discussão sobre operações, estruturas de dados e implementações genéricas de conjuntos. Em seguida, analisamos o caso especial da União de Conjuntos Disjuntos, que consiste em realizar dois tipos de operações sobre uma coleção de conjuntos disjuntos inicialmente unitários: UNION (A, B, C) e FIND (x). Uma sequência de até n − 1 UNIONs e m FINDs intercalados aleatoriamente pode ser executada em tempo $O((m + n).\alpha(m + n, n))$, onde α está relacionado ao inverso da função de Ackermann. Posteriormente, é apresentado o problema para o caso particular em que a árvore de uniões é conhecida *a priori*, sendo a sequência executada em tempo $O(m + n)$, neste caso. Mostramos uma aplicação em um problema relacionado a digrafos: o reconhecimento de digrafos de fluxo redutíveis, que nos proporciona uma comparação entre os *algoritmos alpha* e o algoritmo linear para o caso especial da união estática. Finalizamos comentando mais algumas aplicações para as operações UNION e FIND.

Abstract of Thesis presented to COPPE/UFRJ as partial fulfillment of the requirements for the degree of Master of Science (M. Sc.).

COLLECTIONS OF DISJOINT SETS: OPERATIONS AND ALGORITHMS

Oswaldo Vernet de Souza Pires

April, 1990

Thesis Supervisor: LILIAN MARKENZON

Department: COMPUTING AND SYSTEMS ENGINEERING

This work discusses at first the operations, data structures and generic implementations of sets. Following we analyse the special case of Disjoint Set Union which consists in performing two kinds of operations on a collection of initially single sets: UNION (A, B, C) and FIND (x). A sequence of up to n - 1 UNIONs and m FINDs ramdomly intermixed may be executed in O ((m + n).α (m + n, n)) time, where α is related to the inverse of Ackermann's function. It will be also presented the problem for the special case in which the union tree is known in advance; in this case, the sequence may be performed in O (m + n) time. An application is shown in a problem concerning digraphs: testing flow graph reducibility, which allows us to compare *alpha algorithms* against the linear one. Finally some further applications for the operations are commented.

ÍNDICE DOS CAPÍTULOS

CAPÍTULO I

INTRODUÇÃO

A presente monografia trata essencialmente da implementação e análise de algoritmos que manipulam coleções de conjuntos disjuntos através das operações mencionadas na literatura como UNION e FIND. Este trabalho tem três objetivos. De início, desejamos reunir pela primeira vez em um só texto diversos resultados obtidos ao longo dos anos acerca destes algoritmos. Outro objetivo é dar caráter didático aos conceitos, análises e resultados apresentados, por vezes obscuros na literatura consultada, impondo-lhes uma sequência lógica e uma abordagem mais detalhada e acessível. Por fim, procuramos incluir no texto todos os algoritmos em um pseudo-código próximo a Pascal, simplificando a tarefa de quem porventura venha a implementá-los.

O texto encontra-se dividido em cinco capítulos. Neste primeiro capítulo de introdução, abordamos a idéia genérica de conjunto, propondo uma abstração para o *tipo conjunto* e mostrando duas implementações mais comuns empregadas pelas linguagens de programação que suportam tal construto, como Pascal. Ainda neste capítulo motivamos, com um problema acerca de relações de equivalência, o uso de florestas direcionadas na representação computacional de coleções de conjuntos disjuntos; decorrem então claramente as operações UNION e FIND, motivo central desta monografia.

Em um segundo momento, preocupamo-nos definitivamente com estas operações. Analisamos, de início, uma generalização algébrica do problema, da qual surgem as primitivas LINK, EVAL e UPDATE e as idéias de compressão de caminhos e união por tamanho. As operações UNION e FIND tornam-se, por conseguinte, casos particulares das anteriores. O laborioso cálculo da complexidade de tempo para execução de uma sequência de UNIONs e FINDs é então apresentado. Neste ponto

em particular foram estudadas várias publicações nas quais esta análise evolui até uma versão bastante concisa; o trabalho consistiu então em obter uma demonstração que conservasse a generalidade conseguida em algumas versões aliada ao cálculo da complexidade com amortização, que só aparece por explícito em um trabalho razoavelmente recente. Ainda nesta sequência, incluímos comentários relevantes acerca do limite inferior para o problema e de técnicas alternativas à compressão e união por tamanho.

A existência de um algoritmo linear para um caso particular do problema é examinada, de forma didática, no Capítulo III.

O Capítulo IV aborda uma aplicação para as operações UNION e FIND: o reconhecimento de digrafos de fluxo redutíveis. Esta classe de digrafos tem papel de relevo em técnicas de otimização de código utilizadas em compiladores. Esta aplicação, cuidadosamente implementada, proporciona-nos uma nítida comparação entre os *algoritmos alpha* e o algoritmo linear apresentado no Capítulo III.

Finalizamos citando algumas aplicações para os algoritmos estudados e discutindo os resultados obtidos para os testes que foram realizados.

No restante deste trabalho consideramos conhecidos conceitos introdutórios de Teoria dos Grafos, Análise de Algoritmos e Estruturas de Dados. Todos os logaritmos mencionados no texto, exceto quando a base for explicitada, são de base 2.

I.1. O Tipo Abstrato Conjunto.

Como mencionam HOROWITZ & SAHNI [7], um *Tipo Abstrato de Dados (TAD)* é uma tripla da forma $\langle V, O, A \rangle$, onde V é um conjunto de Domínios, O é um conjunto de Operações (funções) que envolvem elementos pertencentes aos domínios definidos

em D e A é um conjunto de Axiomas que definem a semântica das operações de O.

Assim sendo, o Tipo Abstrato de Dados Conjunto pode ser definido pela tripla ⟨V, O, A⟩, sendo V, O e A definidos como se segue.

V = (C, E, B, ℕ)
onde:

E = Domínio Fundamental, de onde são obtidos os elementos para constituir os Conjuntos nos moldes do Tipo Abstrato;
C = Domínio de Todos os Conjuntos que podem ser formados com elementos de E;
B = O Domínio Booleano (FALSO, VERDADEIRO);
ℕ = O Domínio dos Números Naturais;

O = (VAZIO, é_VAZIO, CONS, PERTENCE, INCLUI, EXCLUI, CARD)
onde:

VAZIO :	→ C	Representa o conjunto vazio;
é_VAZIO:	C → B	Dado um conjunto, verifica se ele é vazio;
CONS:	C x E → C	Dados um conjunto e um elemento, acrescenta o elemento ao conjunto (esta operação é chamada *construtor* do tipo abstrato e apenas serve de apoio às demais);
PERTENCE:	C x E → B	Dados um conjunto e um elemento, verifica se o elemento pertence ao conjunto;
INCLUI:	C x E → C	Dados um conjunto e um elemento, acrescenta o elemento ao conjunto, se ele não pertencer ao conjunto;
EXCLUI:	C x E → C	Dados um conjunto e um elemento, retira o elemento do conjunto, se ele pertencer ao conjunto;

CARD: C -> IN Dado um conjunto, informa quantos elementos ele possui, ou seja, sua cardinalidade.

A = { (1), (2), (3), (4), (5), (6), (7), (8), (9) }

Sejam: c ∈ C; e, x ∈ E.

(1) é_VAZIO (VAZIO) = VERDADEIRO
(2) é_VAZIO (CONS (c, e)) = FALSO
(3) PERTENCE (VAZIO, e) = FALSO
(4) PERTENCE (CONS (c, e), x) =
Se x = e *então*
VERDADEIRO
Caso contrário
PERTENCE (c, x)
(5) INCLUI (c, e) =
Se PERTENCE (c, e) *então*
c
Caso contrário
CONS (c, e)
(6) EXCLUI (VAZIO, x) = VAZIO
(7) EXCLUI (CONS (c, e), x) =
Se x = e *então*
c
Caso contrário
CONS (EXCLUI (c, x), e)
(8) CARD (VAZIO) = 0
(9) CARD (CONS (c, e)) = 1 + CARD (c)

Para exemplificar a manipulação de conjuntos através do Tipo Abstrato, vamos considerar a instância em que E = IN. Devemos ter em mente que, sendo CONS o construtor do tipo, o conjunto A = { 1, 2, 3, 4 }, reescrito como ((({} U { 1 }) U { 2 }) U { 3 }) U { 4 }, pode ser representado pela sucessiva aplicação de CONS: CONS (CONS (CONS (CONS (VAZIO, 1), 2), 3), 4). O resultado de A - { 1 } é obtido fazendo-se

EXCLUI (CONS (CONS (CONS (CONS (VAZIO, 1), 2), 3), 4), 1)

Usando (7) com c = CONS (CONS (CONS (VAZIO, 1), 2), 3), e = 4, x = 1:

CONS (EXCLUI (CONS (CONS (CONS (VAZIO, 1), 2), 3), 1), 4)

Usando (7) com c = CONS (CONS (VAZIO, 1), 2), e = 3, x = 1:

CONS (CONS (EXCLUI (CONS (CONS (VAZIO, 1), 2), 1), 3), 4)

Usando (7) com c = CONS (VAZIO, 1), e = 2, x = 1:

CONS (CONS (CONS (EXCLUI (CONS (VAZIO, 1), 1), 2), 3), 4)

Usando (7) com c = CONS (VAZIO, 1), e = 1, x = 1:

CONS (CONS (CONS (VAZIO, 2), 3), 4)

que é equivalente a (({} U { 2 }) U { 3 }) U { 4 } = { 2, 3, 4 }

A cardinalidade de A é obtida com

CARD (CONS (CONS (CONS (CONS (VAZIO, 1), 2), 3), 4))

Usando (9) com c = CONS (CONS (CONS (VAZIO, 1), 2), 3), e = 4:

1 + CARD (CONS (CONS (CONS (VAZIO, 1), 2), 3))

Usando (9) com c = CONS (CONS (VAZIO, 1), 2), e = 3:

1 + 1 + CARD (CONS (CONS (VAZIO, 1), 2))

Usando (9) com c = CONS (VAZIO, 1), e = 2:

1 + 1 + 1 + CARD (CONS (VAZIO, 1))

Usando (9) com c = VAZIO, e = 1:

1 + 1 + 1 + 1 + CARD (VAZIO)

Usando (8):

1 + 1 + 1 + 1 + 0 = 4.

O principal objetivo ao definirmos abstratamente um tipo é isolarmos a manipulação das estruturas daquele tipo da implementação específica que queiramos dar às operações. Isto é particularmente útil para o encapsulamento das estruturas, tornando disponível ao usuário apenas o conjunto de rotinas que implementam de maneira transparente as operações, sendo portanto indiferente a forma pela qual a implementação foi realizada. Para o Tipo Abstrato Conjunto, existem duas implementações mais comuns que mencionaremos nas próximas seções.

I.2. Implementação com Vetores de Bites.

A primeira implementação proposta comumente para o Tipo Abstrato Conjunto utiliza um vetor de bites indicando ausência ou presença de elementos. Seja o conjunto K a ser representado; evidentemente, K deve ser finito. Estabelecemos, de início, uma enumeração injetora para os elementos de K, ou seja, uma função injetora E : K -> ℕ, que associa um número a cada elemento do conjunto. Sejam

$$a = \min_{x \in K} \{ E(x) \}, \qquad \text{(o menor número associado)}$$

$$b = \max_{x \in K} \{ E(x) \}, \qquad \text{(o maior número associado)}$$

Desta forma, ao invés de lidarmos com os elementos de K, lidamos com suas imagens via E, ou seja, com os números naturais a eles atribuídos. Utilizando um bite para indicar a presença de cada elemento, precisaremos ao todo de b - a + 1 bites para representar K, perfazendo um total de ⌈ (b - a + 1) / 8 ⌉ octetos. Evidentemente, a escolha de E deve ser tal que minimize b - a + 1, sendo ideal a enumeração em que b - a + 1 = | K |.

O algoritmo (I.1) mostra a implementação do Tipo Abstrato de Dados Conjunto utilizando vetor de bites. O operador & significa "and" bite a bite, | significa "or" bite a bite, "~ a" significa complemento a 1 de a e "a << b" significa a * 2^b, ou seja a deslocado b vezes para a esquerda. "div" e "mod" representam divisão inteira e resto da divisão, respectivamente.

Algoritmo I.1: Implementação de Conjuntos com Vetores de Bites.

```
Tipos
    CONJ = estrutura
               conj : arranjo [0 .. ?] de octetos;
               min, max, card, noctetos : inteiros
           fim;

proc VAZIO (Ref C : CONJ; a, b : inteiros);
var
    i : inteiro;
início
    C.min := a;
    C.max := b;
    C.noctetos := (b - a + 8) div 8;
    Aloca C.noctetos octetos para a variável C.conj;
    Para i := 0 até C.noctetos - 1 faça
        C.conj [i] := 0;
    C.card := 0
fim;
```

```
func É_VAZIO (C : CONJ) : lógica;
início
    Retorne (C.card = 0)
fim;

func PERTENCE (C : CONJ; x : inteiro) : lógica;
início
    Se x < C.min ou x > C.max então
        Retorne (FALSO);
    x := x - C.min;
    Se bite (C.conj [x div 8], x mod 8) = 1 então
        Retorne (VERDADEIRO)
    Caso contrário
        Retorne (FALSO)
fim;

proc INCLUI (Ref C : CONJ; x : inteiro);
início
    Se x < C.min ou x > C.max então
        Retorne;
    x := x - C.min;
    Se bite (C.conj [x div 8], x mod 8) = 0 então
        Liga_bite (C.conj [x div 8], x mod 8);
        C.card := C.card + 1
    fim
fim;

proc EXCLUI (Ref C : CONJ; x : inteiro);
início
    Se x < C.min ou x > C.max então
        Retorne;
    x := x - C.min;
    Se bite (C.conj [x div 8], x mod 8) = 1 então
        Desliga_bite (C.conj [x div 8], x mod 8);
        C.card := C.card - 1
    fim
fim;
```

```
func CARD (C: CONJ) : inteiro;
início
    Retorne (C.card)
fim;

func bite (x : octeto; y : inteiro) : inteiro;
início
    Retorne ((x & (1 << y)) ≠ 0)
fim;

proc Liga_bite (Ref x : octeto; y : inteiro);
início
    x := x | (1 << y)
fim;

proc Desliga_bite (Ref x : octeto; y : inteiro);
início
    x := x & ~(1 << y)
fim;
```

As operações é_VAZIO, PERTENCE, INCLUI e EXCLUI levam tempo O (1). Somente a operação VAZIO leva tempo O (b - a + 1). A complexidade de espaço também é O (b - a + 1).

I.3. Implementação com Listas Encadeadas.

O uso de listas encadeadas também se aplica à implementação do Tipo Conjunto. Neste caso, não existe a necessidade de conhecermos previamente a cardinalidade máxima (valor b - a + 1 calculado anteriormente), uma vez que listas são estruturas dinâmicas e o limite decorrerá somente da disponibilidade de memória principal para os algoritmos. Com o uso de listas, torna-se desnecessário enumerar os elementos do conjunto, podendo estes serem armazenados nas próprias células que compõem a lista diretamente. Deve-se

ter o cuidado de não incluir um elemento mais de uma vez na lista, para evitar repetições e, consequentemente, o cômputo incorreto da cardinalidade. O algoritmo (I.2) mostra a implementação. A constante NULL indica fim de lista.

Algoritmo I.2: Implementação de Conjuntos com Listas Encadeadas.

```
Tipos
    T = ..........;     /* Tipo-Base dos Elementos */
    CONJ = estrutura
                conj : lista de T;
                card : inteiro
           fim;

proc VAZIO (Ref C : CONJ);
início
    C.conj := lista_vazia;
    C.card := 0
fim;

func É_VAZIO (C : CONJ) : lógica;
início
    Retorne (C.card = 0)
fim;

func PERTENCE (C : CONJ; x : T) : lógica;
var
    y : T;
início
    y := Primeiro Elemento de C.conj;
    Enquanto y ≠ NULL faça
        Se y = x então
            Retorne (VERDADEIRO);
        y := Próximo Elemento de C.conj
    fim;
    Retorne (FALSO)
fim;
```

```
proc INCLUI (Ref C : CONJ; x : T);
var
    y : T;
início
    y := Primeiro Elemento de C.conj;
    Enquanto y ≠ NULL faça
        Se y = x então
            Retorne;
        y := Próximo Elemento de C.conj
    fim;
    Inclua y no início da lista C.conj;
    C.card := C.card + 1
fim;

proc EXCLUI (Ref C : CONJ; x : T);
var
    y : T;
início
    y := Primeiro Elemento de C.conj;
    Enquanto y ≠ NULL faça
        Se y = x então
            Retire x da lista C.conj;
            C.card := C.card - 1;
            Retorne
        fim;
        y := Próximo Elemento de C.conj
    fim
fim;

func CARD (C: CONJ) : inteiro;
início
    Retorne (C.card)
fim;
```

Com a implementação do algoritmo (I.2), notamos que as operações PERTENCE, INCLUI e EXCLUI levam tempo proporcional à cardinalidade do conjunto; as demais operações tomam tempo

O (i). Quanto ao espaço ocupado por um conjunto, ele é também proporcional à sua cardinalidade. A figura (I.1) mostra um resumo comparativo das duas implementações estudadas.

	Vetor de Bites	Listas Encadeadas
VAZIO	O (b - a + 1)	O (1)
É_VAZIO	O (1)	O (1)
INCLUI	O (1)	O (card)
EXCLUI	O (1)	O (card)
PERTENCE	O (1)	O (card)
CARD	O (1)	O (1)
ESPAÇO	O (b - a + 1)	O (card)

Comparação entre Implementações
Figura I.1

A implementação com listas, aparentemente desvantajosa, pode ser interessante quando a cardinalidade do conjunto se mantiver muito menor do que a cardinalidade máxima, podendo haver economia de espaço. Além disso, duas operações às vezes bastante úteis, que não incluímos no repertório do tipo abstrato, são: o percurso do conjunto aplicando uma função aos elementos presentes, gerando como resultado um novo conjunto constituído pelas imagens destas aplicações (em LISP, esta operação é denominada MAPCAR) e a escolha de um elemento qualquer que pertença ao conjunto. Com vetores de bites, estas operações levam ambas tempo O (b - a + 1) no pior caso, sendo superadas pela implementação com listas, levando tempos O (card) e O (1), respectivamente.

Para conjuntos totalmente ordenados, manter ordenada a lista não altera em complexidade os algoritmos PERTENCE, INCLUI e

EXCLUI no pior caso, mas pode agilizar sobremaneira o caso médio, uma vez que a lista não mais precisará ser integralmente percorrida à procura de um elemento.

I.4. Coleções de Conjuntos Disjuntos.

Como motivação ao desenvolvimento de algoritmos para manipular uma coleção de conjuntos disjuntos, vamos examinar um problema que nos levará ao uso de florestas para representar as coleções bem como às operações mais usuais sobre elas. Reveremos, inicialmente, alguns conceitos da Álgebra.

Definição I.1. Seja um conjunto D e uma relação binária R sobre D, ou seja, $R \subseteq D \times D$. Dizemos que R é uma *Relação de Equivalência* se satisfizer às três propriedades seguintes:

(i) *Reflexividade*: Para todo $x \in D$, $(x, x) \in R$;
(ii) *Simetria*: Para todo $x, y \in D$, se $(x, y) \in R$ então $(y, x) \in R$;
(iii) *Transitividade*: Para todo $x, y, z \in D$, se $(x, y) \in R$ e $(y, z) \in R$ então $(x, z) \in R$.

Um exemplo de relação de equivalência sobre $\mathbb{N}$ seria $R = \{ (x, y) \in \mathbb{N} \times \mathbb{N} \mid x \bmod 3 = y \bmod 3 \}$.

Definição I.2. Seja R uma relação de equivalência sobre D. Seja $x \in D$. Denominamos *Classe de Equivalência* de x com respeito a R o conjunto $C(x) = \{ y \in D \mid (x, y) \in R \}$.

Pela reflexividade de R, concluímos que, para todo $x \in D$, $x \in C(x)$. Pela simetria, temos que, para todo $x, y \in D$, se

$x \in C(y)$ então $y \in C(x)$, ou seja, $C(x) = C(y)$.

No exemplo anterior, a relação R possui três classes de equivalência, a saber:

$C(0) = \{ 0, 3, 6, \ldots, 3n, \ldots \}$

$C(1) = \{ 1, 4, 7, \ldots, 3n + 1, \ldots \}$

$C(2) = \{ 2, 5, 8, \ldots, 3n + 2, \ldots \}$.

Definição I.3. Seja D um conjunto. Uma *Partição* de D é um conjunto formado por subconjuntos de D não vazios, disjuntos dois a dois e que, todos unidos, reproduzem D.

Por exemplo, se $D = \{ 1, 2, 3, 4, 5 \}$, P_1, P_2 e P_3 definidas abaixo são exemplos de partições de D.

$P_1 = \{ \{ 1, 2, 3, 5 \}, \{ 4 \} \}$

$P_2 = \{ \{ 1, 2, 4 \}, \{ 3, 5 \} \}$

$P_3 = \{ \{ 1, 2, 3, 4, 5 \} \}$.

Lema I.1. Seja R uma relação de equivalência sobre D. Então as classes de equivalência de R formam uma partição de D.

Prova. Para que particionem D, as classes de equivalência de R devem ser não vazias, sua união deve reproduzir D e devem ser disjuntas duas a duas. Os dois primeiros requisitos são facilmente verificáveis, pois, existindo a classe C (x), teremos ao menos $x \in C(x)$; como todo elemento $x \in D$ pertence a uma classe de equivalência C (x), evidentemente a união de todas as classes reproduz D.

Quanto à terceira parte, devemos mostrar que, sendo C (a) e C (b) duas classes de equivalência, se $C(a) \neq C(b)$ então $C(a) \cap C(b) = \{\}$. Ao invés disto, vamos mostrar a contrapositiva: se $C(a) \cap C(b) \neq \{\}$ então $C(a) = C(b)$, ou melhor,

$C(a) \subseteq C(b)$ e $C(b) \subseteq C(a)$.

Vamos mostrar que $C(a) \subseteq C(b)$. Realmente, se $C(a) \cap C(b) \neq \{\}$, então existe $x \in D$ tal que $x \in C(a)$ e $x \in C(b)$, ou melhor, $(x, a) \in R$ e $(x, b) \in R$. Tome $y \in C(a)$. Então $(y, x) \in R$. Como $(x, b) \in R$, pela transitividade, $(y, b) \in R$, ou seja, $y \in C(b)$. Logo, $C(a) \subseteq C(b)$. De modo análogo, provamos que $C(b) \subseteq C(a)$.

Podemos então propor o seguinte problema: seja o conjunto $D = \{ x \in \mathbb{N} \mid 1 \leq x \leq n \}$ e Q uma relação binária qualquer sobre D. Desejamos encontrar uma relação de equivalência R sobre D tal que $Q \subseteq R$ e, para qualquer outra relação de equivalência R' sobre D, se $Q \subseteq R'$ então obrigatoriamente $R \subseteq R'$; em outras palavras, R é a relação de equivalência de menor cardinalidade tal que $Q \subseteq R$.

Exemplo: se $D = \{ 1, 2, 3, 4, 5, 6 \}$
e $Q = \{ (1, 2), (2, 5), (3, 3), (6, 4), (5, 3) \}$
então $R = \{ (1, 1), (2, 2), (3, 3), (4, 4), (5, 5),$
$(6, 6), (1, 2), (2, 1), (2, 5), (5, 2),$
$(6, 4), (4, 6), (1, 3), (3, 1), (1, 5),$
$(5, 1), (2, 3), (3, 2), (3, 5), (5, 3) \}$

Uma variante deste problema de maior interesse computacional seria: dados n, os pares de Q e dois elementos $x, y \in D$, verificar se $(x, y) \in R$ sem computar R explicitamente.

A solução sem o cômputo explícito de R pode ser baseada na determinação das classes de equivalência de R, da seguinte forma: inicialmente, consideramos uma coleção de n conjuntos unitários $\{ \{ j \} \mid 1 \leq j \leq n \}$, sendo que cada conjunto representa uma classe de equivalência sendo formada. A cada par $(u, v) \in Q$ obtido da entrada, localizamos os conjuntos C_u e C_v da coleção tais que $u \in C_u$ e $v \in C_v$. Se $C_u \neq C_v$, calculamos a união $C = C_u \cup C_v$, retiramos os conjuntos C_u e

C_v da coleção e acrescentamos C a ela. Processados todos os pares de Q, verificar se (x, y) $\in$ R é verificar se x e y pertencem à mesma classe de equivalência, ou seja, a um mesmo conjunto da coleção final. O algoritmo (I.3) resume esta idéia.

Algoritmo I.3: Solução do Problema da Equivalência.

```
var
    Coleção : conjunto de conjuntos de inteiros;
    C_u, C_v : conjuntos de inteiros;
    i, n, x, y, u, v : inteiros;

início
    Coleção := ();
    Para i := 1 até n faça
        Coleção := Coleção U { { i } };
    Para (u, v) ∈ Q faça
        C_u := Localiza (u);
        C_v := Localiza (v);
        Se C_u ≠ C_v então
            Coleção := Coleção - { C_u } - { C_v };
            Coleção := Coleção U { C_u U C_v }
        fim
    fim;
    Se Localiza (x) = Localiza (y) então
        Imprima SIM
    Caso contrário
        Imprima NÃO
fim;
```

Para representar a coleção de conjuntos disjuntos, GALLER & FISHER [4] utilizam uma floresta direcionada onde os vértices são rotulados com elementos de D. Cada árvore da floresta representa um conjunto da coleção. Inicialmente, portanto, a floresta é constituída de n vértices (árvores triviais), cada um representando um elemento de D. Duas

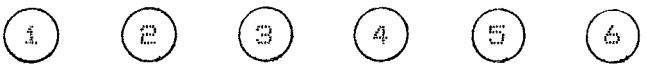
operações são definidas sobre a floresta:

LOCALIZA (v): Determina o conjunto que contém { v }, individualizado pelo rótulo da raiz da árvore onde v se situa;

UNE (u, v): Aglutina os conjuntos que contêm { u } e { v }, combinando as árvores que os representam em uma só.

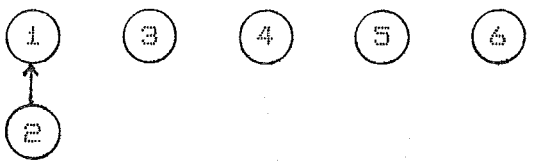
Se representarmos as árvores da floresta como *in-trees*, direcionando as arestas de filho para pai, a operação LOCALIZA (v) resume-se em percorrer o caminho $v \rightarrow x_1 \rightarrow \ldots \rightarrow x_j \rightarrow R_v$, onde R_v é a raiz da árvore que contém { v }. A operação UNE (u, v) consistirá em determinar as raizes das árvores R_u e R_v através de LOCALIZA e acrescentar a aresta (R_u, R_v) ou (R_v, R_u), se $R_u \neq R_v$.

As figuras de (I.2) a (I.6) mostram as alterações na floresta para o exemplo dado. Observe que, ao processarmos o par (3, 3) nada é alterado, pois os vértices pertencem à mesma árvore. Processados os pares, para saber se $(3, 4) \in R$, executamos LOCALIZA (3) = 1 e LOCALIZA (4) = 6, descobrindo que $(3, 4) \notin R$.



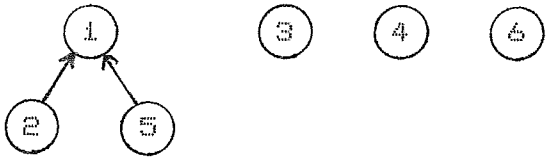
Floresta Inicial

Figura I.2

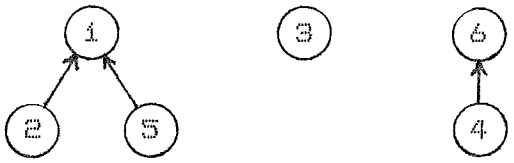


Processamento do par (1, 2)

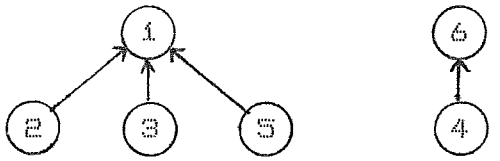
Figura I.3



Processamento do par (2, 5)
Figura I.4



Processamento do par (6, 4)
Figura I.5



Processamento do par (5, 3)
Figura I.6

I.5. Um Breve Histórico.

No restante desta monografia, dedicamo-nos ao estudo dos algoritmos até agora desenvolvidos para implementar eficientemente as operações LOCALIZA e UNE. A referência mais remota que versa sobre tais algoritmos é o artigo de GALLER & FISHER [4] de 1964, no qual é apresentado um algoritmo para processamento de declarações EQUIVALENCE em um programa FORTRAN, já utilizando florestas direcionadas para representar os conjuntos. Entretanto, a análise definitiva da complexidade exata para a execução de uma sequência de operações LOCALIZA e UNE só foi concretizada

por TARJAN [11] em 1975, após algumas publicações - notavelmente a de HOPCROFT & ULLMAN [6] - haverem esboçado limites superiores cada vez mais próximos da complexidade exata.

Data de 1979 o artigo de Tarjan que estabelece o limite inferior para o problema, baseado em um modelo computacional por ele denominado *pointer machine* [12]. A prova apresentada neste artigo continha um erro, apontado posteriormente por BANACHOWSKI [2] em 1980 que, felizmente, não a invalidava. A determinação do limite inferior permitiu constatar a otimalidade dos algoritmos analisados em [11]. Ainda em 1979, VAN LEEUWEN & VAN DER WEIDE [17] relatam uma série de técnicas alternativas para compressão de caminhos, retomadas posteriormente em criteriosa análise por TARJAN & VAN LEEUWEN [15].

Finalmente, à luz do cálculo amortizado da complexidade, reaparece em 1983 [14], com estilo surpreendentemente conciso, a primeira análise feita em 1975.

Um caso particular do problema, estudado por TARJAN & GABOW [3] em 1983, admite um algoritmo linear como solução. Merece também especial destaque o trabalho de TARJAN [13] que trata de uma formalização algébrica do problema, definindo três operações das quais LOCALIZA e UNE tornam-se um caso particular.

CAPÍTULO II

UNIÃO DE CONJUNTOS DISJUNTOS

Neste capítulo, definimos o problema da União de Conjuntos Disjuntos e exploramos alguns algoritmos para resolvê-lo. Inicialmente, abordamos as operações LINK, EVAL e UPDATE, introduzindo as idéias de Compressão de Caminhos e LINK por tamanho. Definimos, a seguir, as operações UNION e FIND, como caso particular das operações LINK-EVAL-UPDATE e examinamos os denominados *algoritmos alpha* para implementá-las.

II.1. Preliminares de Álgebra.

Recordaremos, inicialmente, alguns conceitos básicos de álgebra para a definição das operações LINK-EVAL-UPDATE.

Definição II.1. Seja S um conjunto fechado sob a operação binária $\otimes$, ou seja, $\otimes: S \times S \rightarrow S$. Dizemos que o par $(S, \otimes)$ é um *semigrupo* se $\otimes$ é associativa, isto é, para todo $x, y, z \in S$

$$(x \otimes y) \otimes z = x \otimes (y \otimes z)$$

O conjunto dos números naturais com a operação de soma e o conjunto dos números reais com a operação de multiplicação são exemplos de semigrupos.

Definição II.2. Seja $(S, \otimes)$ um semigrupo. Um elemento $x \in S$ é invertível à direita se existir um elemento x^{-1} tal que, para todo $y \in S$

$$y \circledast x \circledast x^{-1} = y$$

Por exemplo, no semigrupo $(\mathbb{Z}, +)$, todo elemento $x \in \mathbb{Z}$ tem como inverso à direita o seu simétrico $(-x)$.

Vale ressaltar que o inverso à direita, quando existe, não é necessariamente único. Por exemplo, no semigrupo $(S, \circledast)$, sendo $x \circledast y = x$, qualquer $x \in S$ é inverso à direita de qualquer $y \in S$.

Definição II.3. Seja $(S, \circledast)$ um semigrupo. Um elemento $n \in S$ é um nulo à direita se, para todo $x \in S$

$$x \circledast n = n$$

No semigrupo $(\mathbb{Z}, *)$, o número zero é um nulo à direita.

Lema II.1. Seja $(S, \circledast)$ um semigrupo. Se $x, y \in S$ têm inversos à direita, então $x \circledast y$ também o tem.

Prova. Tome $z \in S$ qualquer e sejam x^{-1} e y^{-1} dois inversos à direita de x e y, respectivamente.

$$z \circledast (x \circledast y) \circledast (y^{-1} \circledast x^{-1}) =$$
$$z \circledast x \circledast (y \circledast y^{-1}) \circledast x^{-1} =$$
$$z \circledast (x \circledast x^{-1}) = z.$$

Logo, $y^{-1} \circledast x^{-1}$ é um inverso à direita de $x \circledast y$.

Lema II.2. Seja $(S, \circledast)$ um semigrupo em que os elementos ou são invertíveis à direita ou são nulos à direita. Então, se x^{-1} é um inverso à direita de x, x^{-1} também é invertível à direita.

Prova. Suponha, por absurdo, que x^{-1} não tenha inverso à direita. Logo, x^{-1} é um nulo à direita e $x \circledast x^{-1} = x^{-1}$. Então $x \circledast x \circledast x^{-1} = x^{-1}$, o que contradiz a hipótese de x^{-1} ser um inverso à direita

de x. Portanto, x^{-1} é invertível à direita.

II.2. As Operações LINK-EVAL-UPDATE.

Seja (S, ⊗) um semigrupo e denominemos a operação ⊗ de *multiplicação* ou *produto*. Suponhamos S finito, com n elementos, e seja F uma floresta direcionada de n vértices rotulados com elementos de S, inicialmente constituída de n árvores triviais (com um vértice cada). Estamos interessados em algoritmos para computar uma sequência de três tipos de operações sobre a floresta F, definidas em [13]:

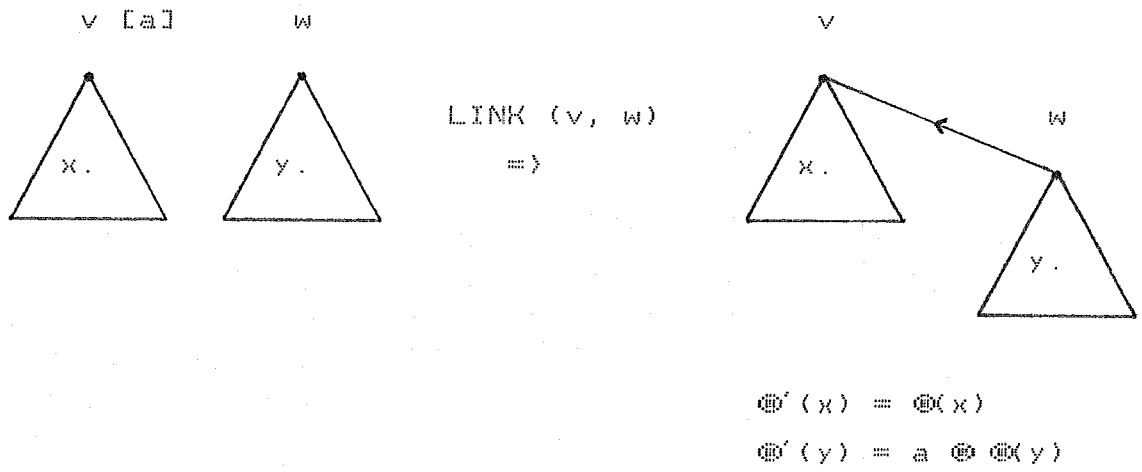
(i) EVAL (x) — Determina a raiz r da árvore que contém o vértice x e retorna como resultado o produto de todos os rótulos no caminho de x até r na ordem contrária. Este produto é denominado *valor* do vértice x, representado por ⊗(r, x) ou ⊗(x), simplesmente.

(ii) LINK (v, w) — Combina as árvores cujas raizes são v e w em uma única árvore mediante o acréscimo da aresta (w, v) à floresta, tornando v sucessor de w. A raiz da árvore resultante será, portanto, v.

(iii) UPDATE (r, x) — Sendo r a raiz de uma árvore com rótulo a, troque o rótulo de r por x ⊗ a.

Podemos, a partir da definição, observar os seguintes resultados:

a. As árvores da floresta têm arestas direcionadas de filho para pai, sendo, portanto, *in-trees*. Assim, os termos *filho* e *predecessor*, bem como *pai* e *sucessor*, são sinônimos.



Efeitos das Operações LINK e UPDATE
Figura II.1

b. Após uma operação LINK (v, w), os valores dos vértices da antiga árvore cuja raiz era v não se alteram, ao passo que os da antiga árvore enraizada em w ficam multiplicados pelo rótulo de v à esquerda.

c. A operação UPDATE (r, x) tem como efeito multiplicar os valores de todos os vértices da árvore enraizada em r por x.

d. Nada é suposto quanto à comutatividade de @. Logo, não é irrelevante a ordem dos vértices no caminho durante uma operação EVAL.

Estas observações estão ilustradas na figura (II.1), onde vértices são representados por letras minúsculas e o rótulo de um vértice é colocado entre colchetes; $\oplus'(v)$ simboliza o valor de v após a operação e $\oplus(v)$, o valor de v anterior a ela.

II.3. Implementação das Operações LINK-EVAL-UPDATE.

Uma maneira bastante trivial de implementar as operações LINK-EVAL-UPDATE é utilizar dois arranjos unidimensionais de n elementos, denominados *pai* e *rot*. Para todo vértice $v \in F$, pai [v] é o vértice pai de v em F e rot [v] é o rótulo atual atribuído a v. Se v é raiz de uma árvore, fazemos pai [v] = 0. Inicialmente, para todo $v \in F$, pai [v] = 0 (todos os vértices são raizes) e rot [v] = rótulo inicial de v.

Algoritmo II.1: Implementação das Operações LINK, EVAL e UPDATE.

```
Tipos
    T = .......;  /* Tipo-Base dos Rótulos */

var
    pai : arranjo [1..n] de inteiros;
    rot : arranjo [1..n] de T;

proc inicializa;
var
    i : inteiro;
início
    Para i := 1 até n faça
        pai [i] := 0;
        rot [i] := "rótulo inicial de i"
    fim
fim;
```

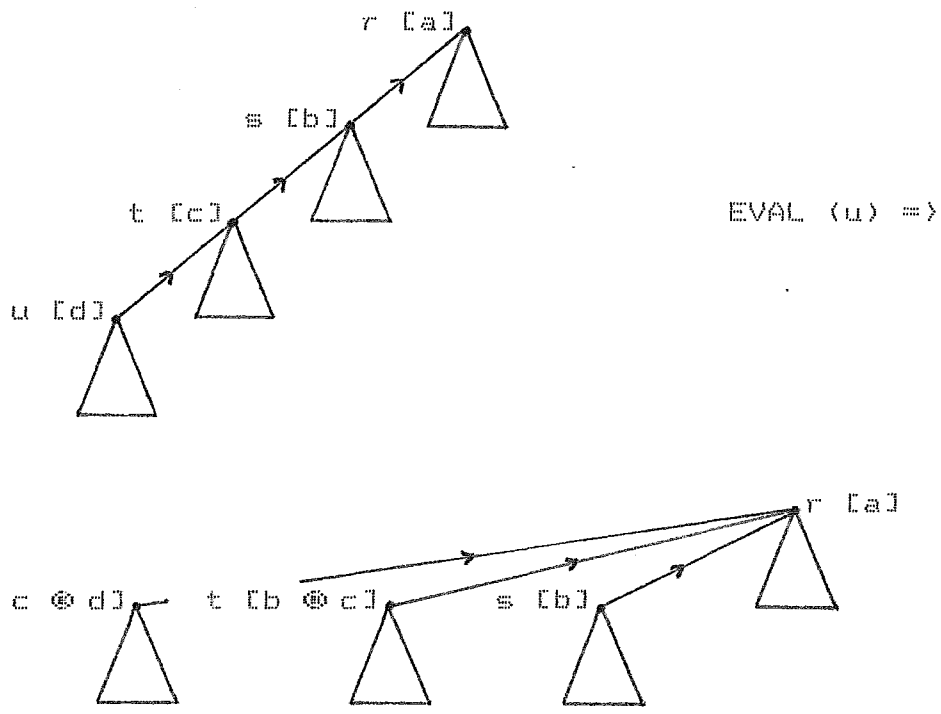
```
proc LINK (v, w : inteiros);
início
    Se pai [v] ≠ 0 ou pai [w] ≠ 0 então
        Erro (v, " ou", w, " não são raizes")
    Caso contrário
        pai [w] := v
fim;

func EVAL (v : inteiro): T;
início
    Se pai [v] = 0 então     /* v é raiz */
        Retorne (rot [v])
    Caso contrário
        Retorne (EVAL (pai [v]) @ rot [v])
fim;

proc UPDATE (r : inteiro; x : T);
início
    Se pai [r] ≠ 0 então
        Erro (r, "nao é raiz")
    Caso contrário
        rot [r] := x @ rot [r]
fim;
```

Conforme pode ser observado, LINK e UPDATE são executadas em tempo O (1). EVAL (v), entretanto, toma tempo proporcional ao número de vértices no caminho de v à raiz da árvore que o contém. Se imaginarmos uma sequência de n - 1 operações LINK que transformem a floresta em uma árvore degenerada (linha reta), EVAL (v) levará tempo O (n) no pior caso e uma sequência de m operações EVAL poderá perfazer O (m.n), o que não é um resultado muito eficiente.

A técnica utilizada para obter uma primeira melhoria de eficiência baseia-se na idéia de Compressão de Caminhos, introduzida por GALLER & FISHER [4] e formalizada por TARJAN [11]:

Ao executar uma operação EVAL (v), torne todos os vértices no caminho de v a r predecessores (filhos) de r (exceto o próprio r) e modifique os rótulos convenientemente de forma que seus valores sejam corretos em operações EVAL subsequentes.

A figura (II.2) ilusta este processo.



Compressão de Caminhos
Figura II.2

A floresta assim representada, denominada *floresta virtual*, não mais corresponde fielmente à *floresta real* resultante da sequência de operações LINK. Entretanto, as seguintes relações valem entre as duas:

(i) A cada árvore T da floresta real F corresponde uma árvore T' da floresta virtual F', constituída pelos mesmos vértices eventualmente com diferentes rótulos;

(ii) Árvores correspondentes T e T' possuem a mesma raiz com o mesmo rótulo;

(iii) Os valores dos vértices computados em F e F' são idênticos: $\oplus_F(v) = \oplus_{F'}(v)$.

O algoritmo (II.2) apresenta a nova versão de EVAL usando compressão de caminhos. LINK e UPDATE são implementadas como antes (algoritmo (II.1)).

Algoritmo II.2: EVAL com Compressão de Caminhos.

```
proc COMPRIME (v : inteiro);
início
    Se pai [pai [v]] ≠ 0 então
        COMPRIME (pai [v]);
        rot [v] := rot [pai [v]] ⊕ rot [v];
        pai [v] := pai [pai [v]]
    fim
fim;

func EVAL (v : inteiro) : T;
início
    Se pai [v] = 0 então
        Retorne (rot [v])
    Caso contrário
        COMPRIME (v);
        Retorne (rot [pai [v]] ⊕ rot [v])
    fim
fim;
```

II.4. LINK por Tamanho e Florestas Balanceadas.

Conforme provaremos na seção (II.7), se conseguirmos que a compressão de caminhos se efetue sobre florestas balanceadas, melhoramos substancialmente a complexidade dos algoritmos. Para isto, vejamos a definição de floresta

balanceada.

Definição II.4. Seja F uma floresta de n vértices. Dizemos que F é *balanceada* com respeito às constantes a > 1 e c > 0 se, para todo h, o número de vértices com altura h não excede c * n / a^h.

Recorde-se que a altura de um vértice em uma floresta é o comprimento do maior caminho partindo de uma folha da árvore que o contém até ele; as folhas têm altura 0. Intuitivamente, a idéia de floresta balanceada corresponde ao fato de a maioria dos vértices ter altura pequena.

A técnica que denominamos *LINK por tamanho*, segundo TARJAN [11], consiste no seguinte:

> *Ao combinar duas árvores por uma operação LINK, tornar a raiz da árvore que possui menos vértices predecessor (filha) da raiz da árvore com maior número de vértices.*

Intuitivamente, percebe-se que, com esta cautela, evitamos o surgimento de árvores degeneradas, ou seja, de vértices muito distantes das raizes das árvores que os contêm. Passemos à prova de que a floresta gerada por uma sequência de operações LINK por tamanho é balanceada.

Lema II.3. Em uma floresta gerada por uma sequência de operações LINK por tamanho, uma árvore com altura h tem, no mínimo, 2^h vértices.

Prova. Indução em h. Para h = 0, temos a árvore trivial de um só vértice com altura 0. Fixe h e suponha o resultado válido para valores de alturas inferiores a h. Seja T uma árvore de altura h. Então T foi obtida pela combinação de outras duas árvores T_1 e T_2, sendo, por exemplo, T_1 de altura h - 1 e com menos vértices que T_2. Pela hipótese de indução, T_1 tem no

mínimo 2^{m-1} vértices e, forçosamente, T_w tem também, no mínimo, 2^{m-1} vértices. Logo, T tem 2^m vértices, no mínimo.

Definição II.5. Seja F uma floresta e U uma sequência de operações LINK-EVAL-UPDATE. Seja v um vértice de F. Denominamos *posto* de v com respeito à sequência U à altura de v na floresta resultante da execução apenas das operações LINK de U, denominada *Floresta de Referência*.

Lema II.4. Utilizando-se LINK por tamanho, pode haver no máximo n / 2^m vértices de posto p na floresta de referência.

Prova. Pelo lema (II.3), todo vértice de posto p tem, no máximo, 2^m descendentes na floresta. Como os conjuntos de descendentes de dois vértices de mesma altura são disjuntos e há no máximo n / 2^m conjuntos disjuntos de 2^m ou mais vértices, não pode haver mais de n / 2^m vértices de posto p.

Corolário II.1. Nenhum vértice tem posto maior que log n.

Pelo lema (II.4), concluímos que uma floresta construída por uma sequência de operações LINK por tamanho é balanceada com respeito às constantes c = 1 e a = 2.

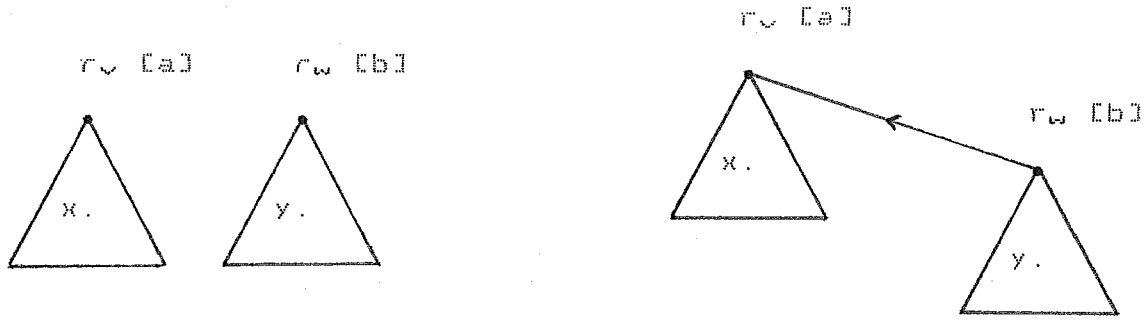
II.5. Semigrupos com Nulos e Inversos à Direita.

Analisaremos um caso particular de semigrupo para o qual os conceitos de compressão de caminhos e balanceamento de florestas aplicam-se em conjunto.

Seja (S, ⊗) um semigrupo para o qual todo elemento x ∈ S ou é um nulo à direita ou possui pelo menos um inverso x^{-R} à direita. Para garantir o balanceamento da floresta, utilizaremos a idéia de LINK por tamanho. Assim, LINK (v, w) acrescenta a aresta (w, v) se a árvore cuja raiz é v tiver mais vértices do que a árvore cuja raiz é w, ou acrescenta a aresta (v, w), caso contrário. Desta maneira, entretanto, não mais garantimos que a raiz da árvore resultante seja v, como é previsto pela definição da operação LINK (v, w), mas operações LINK futuras poderão mencionar v como raiz, causando inconsistências na representação da floresta. Além disso, se a raiz for w, EVAL não mais calculará corretamente os valores dos vértices. Para contornar estes problemas, o algoritmo que implementa LINK (v, w) deve tomar alguns cuidados adicionais (denotamos por rot (v) o rótulo associado ao vértice v):

a. Antes de efetuarmos o LINK propriamente dito, devemos obter as raizes r_v e r_w das árvores T_v e T_w, caso elas não sejam v e w respectivamente.

b. Se T_v tem número de vértices maior ou igual ao de T_w, pela regra de LINK por tamanho, r_w torna-se filho de r_v. Por conseguinte, os valores de todos os vértices que pertenciam a T_w ficam multiplicados à esquerda por rot (r_v), que era o desejado na definição inicial de LINK. Tudo se passa como anteriormente, como mostra a figura (II.3) (⊗'(x) é o valor de x após o LINK e ⊗(x) é o valor de x antes do LINK).



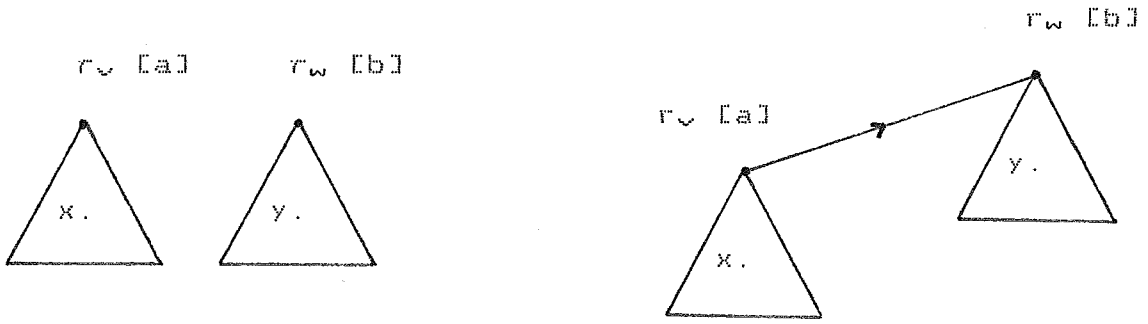
$\circledast'(x) = a \circledast \ldots \circledast \text{rot}\ (x) = \circledast(x)$

$\circledast'(y) = a \circledast b \circledast \ldots \circledast \text{rot}\ (y) = a \circledast \circledast(y)$

LINK (v, w) se $|T_v| \geq |T_w|$

Figura II.3

c. Se T_v tem menos vértices do que T_w, r_v torna-se filho de r_w e o efeito de multiplicação dos vértices de T_w por rot (r_v) não mais é conseguido, como está ilustrado na figura (II.4): o valor de um vértice que pertencia a T_w não fica multiplicado à direita por rot (r_v) como desejávamos, ao passo que o valor de um vértice que pertencia a T_v fica multiplicado por rot (r_w)!



$\circledast'(x) = b \circledast a \circledast \ldots \circledast \text{rot}\ (x) = b \circledast \circledast(x)$

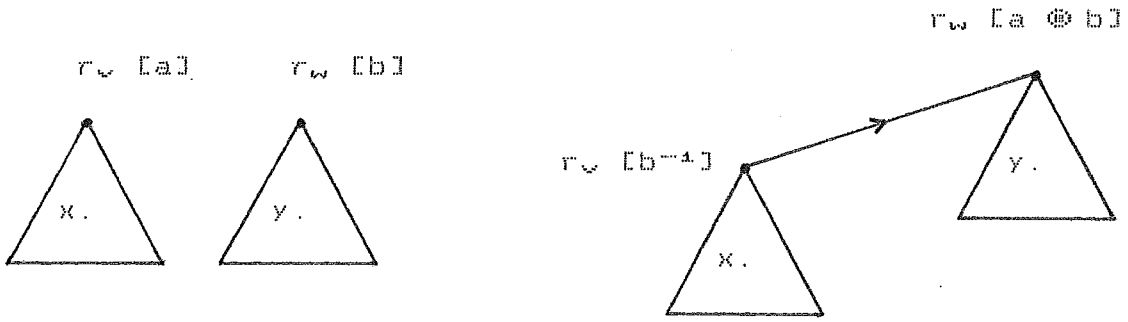
$\circledast'(y) = b \circledast \ldots \circledast \text{rot}\ (y) = \circledast(y)$

Efeito Errôneo de LINK (v, w) se $|T_v| < |T_w|$

Figura II.4

Para corrigir este efeito, vamos supor que rot (r_w) tenha um inverso à direita. Usamos então um artifício: trocamos

o rótulo de r_v por um inverso à direita do rótulo de r_w e, simultaneamente, trocamos o rótulo de r_w pelo produto rot $(r_v) \circledast$ rot (r_w). Com isto, os valores dos vértices que pertenciam a T_v ficam inalterados e os valores dos vértices que pertenciam a T_w ficam multiplicados por rot (r_v), exatamente como desejávamos. Este procedimento é mostrado na figura (II.5), onde b^{-1} é um inverso à direita de b.



$$\circledast'(x) = a \circledast b \circledast b^{-1} \circledast \ldots \circledast \text{rot}\ (x) =$$
$$a \circledast \ldots \circledast \text{rot}\ (x) = \circledast(x)$$
$$\circledast'(y) = a \circledast b \circledast \ldots \text{rot}\ (y) = a \circledast \circledast(y)$$

LINK (v, w) se $|T_v| < |T_w|$

Figura II.5

d. Ainda no caso anterior, se porventura rot (r_w) não possui inverso à direita então, pelo pressuposto inicial, rot (r_w) é um nulo à direita. Isto significa que, para qualquer vértice $y \in T_w$

$$\circledast'(y) = \text{rot}\ (r_v) \circledast \text{rot}\ (r_w) \circledast \ldots \circledast \text{rot}\ (y) =$$
$$= \text{rot}\ (r_w) \circledast \ldots \circledast \text{rot}\ (y) =$$
$$= \circledast(y)$$

ou seja, seu valor permanecerá inalterado, indicando que a operação LINK (v, w) perde seu efeito. Já que w não mais será mencionado como raiz em LINKs futuros, LINK (v, w) não precisa ser realizado.

Estes procedimentos fazem surgir uma floresta virtual, como no algoritmo (II.2), que guarda para com a floresta real relações ligeiramente diferentes das anteriores, a saber:

(i) A cada árvore T da floresta real F corresponde uma árvore T′ da floresta virtual F′, constituída pelos mesmos vértices;

(ii) As raizes de duas árvores correspondentes T e T′ não necessariamente coincidem;

(iii) $\phi_F(v) = \phi_{F'}(v)$.

Mostramos, no algoritmo (II.3), a implementação de LINK usando a regra de tamanho. EVAL e COMPRIME são implementadas exatamente como na seção (II.3). Nesta implementação, acrescentamos os arranjos *tam* e *raiz*, de n posições, tal que tam [v] contém o número de vértices da árvore cuja raiz é v, para todo v que seja raiz de árvore, e raiz [v] permite localizar a raiz da árvore que contém o vértice v. Inicialmente, para todo $v \in F$, tam [v] = 1 e raiz [v] = v (ou seja, todos são raizes).

Algoritmo II.3: LINK por Tamanho.

```
var
    pai, tam, raiz : arranjos [1..n] de inteiros;
    rot            : arranjo  [1..n] de T;

proc inicializa;
var
    j : inteiro;
início
    Para j := 1 até n faça
        pai  [j] := 0;
        tam  [j] := 1;
```

```
            raiz [j] := j;
            rot  [j] := "rótulo inicial de j"
        fim
    fim;

    proc LINK (v, w : inteiros);
    var
        r_v, r_w : inteiros;
        x        : T;
    início
        r_w := raiz [w];
        r_v := raiz [v];
        Se r_v = 0 ou r_w = 0 então
            Erro (v, " ou", w, " não são raizes");
            Retorne
        fim;
        Se r_v = rw então
            Retorne;
        Se rot [r_w]^-1 tem inverso à direita então
            Seja rot [r_w]^-1 um dos inversos;
            Se tam [r_v] ≥ tam [r_w] então
                pai [r_w] := r_v;
                tam [r_v] := tam [r_v] + tam [r_w]
                /* Aqui, raiz [v] não se altera */
            Caso contrário
                pai [r_v] := r_w;
                tam [r_w] := tam [r_v] + tam [r_w];
                x := rot [r_v];
                rot [r_v] := rot [r_w]^-1;
                rot [r_w] := x ® rot [r_w];
                raiz [v] := r_w
            fim
        fim;
        /* Para LINKs futuros, w deixa de ser raiz */
        raiz [w] := 0;
    fim;
```

O procedimento UPDATE exige apenas o cuidado de localizar a raiz da árvore, como é mostrado no algoritmo (II.4).

Algoritmo II.4: Nova Implementação de UPDATE.

```
proc UPDATE (r : inteiro; x : T);
var
    s : inteiro;
início
    s := raiz [r];
    Se s = 0 então
        Erro (r, " não é raiz")
    Caso contrário
        rot [s] := x ⊕ rot [s]
fim;
```

Observamos, finalmente, que a implementação recursiva do procedimento COMPRIME pode ser indesejável em certas situações. Uma forma bastante simples de torná-la iterativa, além do tradicional uso de uma pilha, é seguir os ponteiros do arranjo pai de v a r, invertendo suas direções na ida; posteriormente, seguimo-los de volta, comprimindo os caminhos e atualizando os rótulos (algoritmo (II.5)).

Algoritmo II.5: COMPRIME sem Recursividade.

```
proc COMPRIME (v : inteiro);
var
    x, k : inteiro;
    val  : T;
início
    x := v;
    k := 0;
    Enquanto pai [x] ≠ 0 faça
        y := pai [x];
        pai [x] := k;
```

```
            k := x;
            x := y
        fim;
        val := rot [k];     /* Assuma rot [0] definido */
        Enquanto k ≠ 0 faça
            y := pai [k];
            pai [k] := x;
            rot [k] := val;
            k := y;
            val := val ⊕ rot [k]
        fim
    fim;
```

II.6. União de Conjuntos Disjuntos.

Seja um domínio D finito, com n elementos, e uma coleção de n conjuntos unitários e distintos, constituídos, cada um, por um elemento de D. Cada conjunto (j) ($i \leq j \leq n$) possui um nome N_j, distinto dos demais. O problema conhecido como *União de Conjuntos Disjuntos* consiste em realizar uma sequência de até n − 1 operações UNION e m operações FIND intercaladas aleatoriamente, assim definidas:

UNION (N_a, N_b, N_c): Promove a união dos conjuntos cujos nomes são N_a e N_b originando um novo conjunto cujo nome é N_c. Os antigos conjuntos de nomes N_a e N_b são retirados da coleção e o novo conjunto N_c é a ela acrescentado. N_c deve ser distinto dos demais nomes da coleção.

FIND (x): Para $x \in D$, obtém N_j, nome do conjunto que contém (x), $1 \leq j \leq n$.

Em face desta definição, podemos observar os seguintes fatos:

a. Se a sequência contiver exatamente n − 1 operações UNION, a coleção original de n conjuntos é transformada em uma coleção final constituída por um só conjunto: o próprio D.

b. Cada coleção obtida através de um UNION é uma partição de D. Com efeito, a coleção inicial é uma partição; a cada passo, substituímos dois conjuntos quaisquer da coleção por sua união, que é disjunta dos demais e, unida a eles, reproduz D.

c. Sem perda de generalidade, os nomes dos conjuntos podem ser os próprios elementos de D. Inicialmente, para todo $x \in D$, nome ({ x }) = x. Geralmente, em uma operação UNION (A, B, C), temos C = A ou C = B, garantindo que dois conjuntos terão sempre nomes distintos na sequência de coleções geradas.

d. Uma coleção pode ser representada (não univocamente) por uma floresta de n vértices, sendo cada conjunto da coleção dado por uma árvore. A coleção inicial é, por conseguinte, uma floresta de n árvores triviais. A operação UNION tem como efeito combinar duas árvores da floresta em uma única. Nomes de conjuntos podem ser vistos como rótulos para os vértices da floresta que sejam raizes de árvores.

As observações c e d anteriores estabelecem um forte paralelo entre o problema da União de Conjuntos Disjuntos e as operações LINK-EVAL-UPDATE das seções anteriores. Com efeito, seja o semigrupo (D, ⊛), sendo ⊛ definida assim: para todo $x, y \in D$, $x \circledast y = x$. Em termos das operações LINK-EVAL-UPDATE já conhecidas, UNION e FIND podem ser expressas como se segue:

```
UNION (A, B, A):   LINK (v (A), v (B))
UNION (A, B, B):   LINK (v (A), v (B)); UPDATE (v (A), B)
FIND (x):          EVAL (E (x))
```

onde v (M) corresponde ao vértice que possuía rótulo M inicialmente e E (x) corresponde ao vértice que representa o elemento x ∈ D.

Se levarmos em conta o fato de, no semigrupo (D, ⊛), todo elemento x ∈ D ser um inverso à direita de qualquer elemento y ∈ D, pois para todo x, y, z ∈ D, z ⊛ y ⊛ x = z, estamos diante do caso particular examinado na seção (II.5). Em outras palavras, os algoritmos que implementam LINK-EVAL-UPDATE utilizando compressão de caminhos e LINK por tamanho são aplicáveis ao problema de União de Conjuntos Disjuntos, uma vez que todos os elementos do semigrupo (D, ⊛) são invertíveis à direita. No entanto, dada a simplicidade da operação ⊛ neste caso (que sempre devolve o primeiro termo do produto), o procedimento FIND pode ser reescrito muito mais facilmente que EVAL, como é mostrado no algoritmo (II.6), tendo sido utilizada a versão não-recursiva (algoritmo (II.5)).

Algoritmo II.6: As Operações UNION e FIND.

```
var
    nome, raiz,
    tam, pai     : arranjos [1..n] de inteiros;

proc inicializa;
var
    i : inteiro;
início
    Para i := 1 até n faça
        pai [i] := 0; nome [i] := i;
        tam [i] := 1; raiz [i] := i
    fim
fim;
```

```
func FIND (x : inteiro) : inteiro;
var
    y : inteiro;
início
    y := x;
    Enquanto pai [y] ≠ 0 faça
        y := pai [y];
    /* y é raiz da árvore */
    Enquanto pai [x] ≠ 0 faça
        pai [x] := y;     /* compressão */
        x := pai [x]
    fim;
    Retorne (nome [y])
fim;

proc UNION (A, B, C : inteiros);
var
    r_a, r_b, r_c : inteiros;
início
    r_a := raiz [A]; r_b := raiz [B]; r_c := raiz [C];
    Se r_a = 0 ou r_b = 0 então
        Erro (A, " ou", B, " não são raizes");
        Retorne
    fim;
    Se r_c ≠ r_a e r_c ≠ r_b e r_c ≠ 0 então
        Erro ("Conflito de Nomes na Coleção");
        Retorne
    fim;
    Se r_a = r_b então
        Retorne;
    Se tam [r_a] < tam [r_b] então
        Troque r_a e r_b;
    tam [r_a] := tam [r_a] + tam [r_b];
    pai [r_b] := r_a;
    /* Os nomes A e B deixam de existir */
    raiz [A] := raiz [B] := 0;
    raiz [C] := r_a;        /* Acrescenta C à coleção */
    nome [r_a] := C
fim;
```

Observe que FIND (v) leva tempo proporcional à distância de v à raiz, realizando um duplo percurso do caminho. Visando economia de espaço, podemos armazenar os arranjos *pai* e *tam* em um só arranjo *vet*, fazendo

$$vet[v] = \begin{cases} -tam[v], & \text{se } v \text{ é raiz de uma árvore} \\ pai[v], & \text{caso contrário} \end{cases}$$

Voltando ao problema da equivalência, estudado na seção (I.4), a solução através das operações UNION e FIND é dada no algoritmo (II.7).

Algoritmo II.7: Solução do Problema da Equivalência.

```
proc Equivalência (n, Q, x, y);
var
    u, v, Cu, Cv;
início
    inicializa;
    Para (u, v) ∈ Q faça
        Cu := FIND (u);
        Cv := FIND (v);
        UNION (Cu, Cv, Cu)
    fim;
    Se FIND (x) = FIND (y) então
        Imprima SIM
    Caso contrário
        Imprima NÃO
fim;
```

II.7. Análise de Complexidade.

Vamos abordar o cálculo da complexidade de tempo para a execução de uma sequência de até $n - 1$ UNIONs e m FINDs

sobre um universo de tamanho n, usando compressão de caminhos com e sem união por tamanho. Seguimos a proposta de TARJAN [14], realizando a análise mediante o sistema de créditos e débitos.

A idéia, introduzida por TARJAN [16], consiste em imaginar que o esforço computacional deve ser pago com unidades denominadas *créditos*. Para realizar uma operação, necessitamos de uma determinada quantidade de créditos, que fará a máquina executar por determinado período de tempo. Para cada operação a ser realizada, alocamos então um certo número de créditos. O objetivo é mostrar que todas as operações poderão ser realizadas com a quantidade de créditos prevista. Se porventura acabam os créditos antes que as operações sejam concluídas, podemos tomá-los emprestados, criando *débitos*. Com este esquema, o tempo total para realizar todas as operações é proporcional à soma do número de créditos previstos inicialmente (NC) com o número de débitos acumulados ao fim do processamento (ND).

Para computar então os créditos e débitos envolvidos na execução da sequência de UNIONs e FINDs, vamos utilizar um Sistema de Partições Múltiplas, de forma que cada partição separe os vértices da floresta de acordo com seus postos (veja definição (II.5)). Lembramos desde já que os postos dos vértices são calculados com respeito à floresta de referência e permanecem inalterados durante a execução da sequência, a despeito das compressões realizadas sobre os caminhos. Além disso, no caminho de um vértice v até a raiz de sua árvore, os postos dos vértices crescem estritamente, até o valor máximo $\lfloor \log n \rfloor$ (corolário (II.1)).

Definição II.6. Um *Sistema de Partições Múltiplas (SPM)* é uma quádrupla (k, L, U, B) definindo uma coleção de partições sobre o conjunto $\mathbb{N}$, onde:

(i) $k \in \mathbb{N}$, sendo $k + 1$ o número de partições da coleção;
(ii) L_p é o número de blocos da partição p, $0 \leq p \leq k$.
(iii) $U \in \mathbb{N}$;
(iv) B é uma função que define os limites inferior e superior de cada bloco das partições.

A coleção definida através de um SPM deve satisfazer às seguintes propriedades:

(i) Existem $k + 1$ partições, numeradas de 0 a k.
(ii) Os conjuntos (*blocos*) que constituem a partição p são, ao todo, L_p intervalos da forma bloco (p, q) = $\{ x \in \mathbb{N} : B(p, q) \leq x < B(p, q + 1) \}$, $0 \leq q < L_p$.

(iii) $B(0, q) = q$, $0 \leq q < L_0$.
(iv) $B(p, 0) = 0$, $1 \leq p \leq k$.
(v) $B(p, q) < B(p, q + 1)$, $1 \leq p \leq k$, $0 \leq q < L_p$
(vi) $B(p, L_p) > U$, $0 \leq p \leq k$.

(vii) $L_k = 1$.
(viii) $L_a < L_b$, $1 \leq b < a \leq k$.

A partir desta definição, concluímos imediatamente as seguintes propriedades:

a. A partição 0 é formada por conjuntos unitários.
b. Qualquer natural entre 0 e U pertence a algum bloco da partição p, $0 \leq p \leq k$.
c. A partição k consiste de um só bloco: bloco (k, 0).
d. As partições tornam-se cada vez menos refinadas, de 0 a k, já que o número de blocos decresce estritamente. Como medida do não-refinamento, introduzimos a função R_{pq} como sendo o número de blocos da partição $p - 1$ cuja interseção com bloco (p, q) é não-vazia.

Definição II.7. Dado um SPM, *Nível* de um vértice x, representado por nível (x), é o menor valor

de p para o qual posto (x) e posto (pai (x)) encontram-se em um mesmo bloco da partição p. Se x é raiz de uma árvore, adotamos nível (x) = 0; evidentemente, nos demais casos, $1 \leq$ nível (x) $< k$.

Definição II.8. A *Função de Ackermann* A e seu funcional inverso α são definidos da seguinte forma:

$$A(p, q) = \begin{cases} 2^q; & p = 1,\ q \geq 1 \\ A(p-1, 2); & p \geq 2,\ q = 1 \\ A(p-1, A(p, q-1)); & p \geq 2,\ q \geq 2 \end{cases}$$

$$\alpha(m, n) = \min \{\, p \geq 1 \mid A(p, \lfloor m / n \rfloor) > \log n \,\},$$
para $m \geq n$.

A principal característica da função A é seu crescimento explosivo. Com efeito, temos:

$$A(2, 1) = A(1, 2) = 2^2$$
$$A(2, x) = A(1, A(2, x-1)) = 2^{A(2,\, x-1)} = 2^{2^{\cdots^{2}}}$$
(são, ao todo, x+1 "2"!)

Em decorrência deste fato, $\alpha(m, n)$ é uma função de crescimento extremamente lento. Além disso, à medida que a razão m / n cresce, $\alpha(m, n)$ decresce. Temos ainda

$$A(4, 1) = A(3, 2) = A(2, A(3, 1)) = A(2, A(2, 2)) = A(2, 16)$$

ou seja, para quaisquer valores de m e n, na prática, $\alpha(m, n)$ vale, no máximo, 4.

Para executar a sequência de UNIONs e FINDs, vamos alocar um crédito para cada UNION, já que seu tempo de execução é O (1). Quanto aos FINDs, considere a execução de FIND (v),

sendo $v = x_0, x_1, \ldots, x_{h-1}, x_h = r$ o caminho a ser comprimido, com $x_{j+1} = pai(x_j)$, $0 \leq j < h$. Necessitamos de tantos créditos quantos sejam os vértices no caminho, já que FIND (v) leva tempo proporcional à distância de v a r. Visto não podermos prever de antemão quantos vértices existem de v a r, usamos o seguinte raciocínio: como os níveis dos vértices variam entre 0 e k, vamos alocar um crédito para cada vértice que seja o último de seu nível no caminho, totalizando k + 1 créditos para o FIND; note que esta previsão engloba o caso extremo em que vértices de todos os níveis estejam presentes. Sendo até n − 1 UNIONs e m FINDs, o número de créditos previstos é portanto

$$NC = (n - 1) + m \cdot (k + 1) = n + mk + m - 1$$

Para cada vértice do caminho que não receba crédito, criamos um débito; a tarefa agora consiste em computar o total de débitos ao fim das operações. Para isto, vamos analisar as variações da expressão posto (pai (x_j)) para os diversos vértices x_j do caminho, uma vez que a compressão altera pai (x_j), embora os postos não variem. Consideremos portanto um vértice x_j no caminho que, não sendo o último de seu nível, ocasione um débito. Vamos limitar superiormente o número de débitos associados a x_j enquanto nível (x_j) = p, $1 \leq p < k$. Se nível (x_j) = p então posto (x_j) e posto (pai (x_j)) encontram-se em blocos diferentes da partição (p − 1) e no mesmo bloco q da partição p antes de o FIND ser executado. Como x_j não é o último vértice de nível p no caminho, posto (pai (x_j)) e posto (r) também situam-se em blocos diferentes da partição (p − 1) antes do FIND. Comprimido o caminho, teremos pai (x_j) = r; a compressão modifica o valor da expressão posto (pai (x_j)), fazendo com que posto (pai (x_j)) se desloque de bloco (p − 1, q′) para bloco (p − 1, q"), q" > q′. Isto pode acontecer até no máximo $R_{pq} - 1$ vezes sem que nível (x_j) seja alterado; em outras palavras, x_j pode ocasionar, no máximo, $R_{pq} - 1$ débitos enquanto nível (x_j) = p.

Chamando de N_{pq} o número de vértices com posto pertencente a bloco (p, q), podemos limitar o número de débitos acumulados pelo duplo somatório

$$ND \leq S = \sum_{p=1}^{k} \sum_{q=0}^{L_p - 1} N_{pq} \cdot (R_{pq} - 1)$$

Teorema II.1. Usando compressão de caminhos e união por tamanho, a sequência de até n − 1 UNIONs e m FINDs sobre um universo de n elementos é executada em tempo O ((n + m) . α (m + n, n)).

Prova. Para calcular o duplo somatório S, vamos fixar os parâmetros k, L, U e B de um Sistema de Partições Múltiplas (SPM) adequado para o caso. Assim, como a idéia é particionar os vértices por postos e o valor máximo do posto é log n, tomamos:

$k = \alpha (m + n, n) + 1$

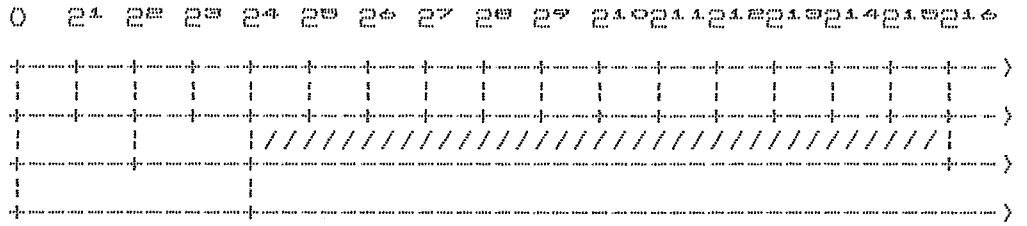
$L_p = \min \{ q \mid A (p, q) > \log n \}$

$U = \lceil \log n \rceil$

$B (p, q) = A (p, q), 1 \leq p < k, 1 \leq q \leq L_p$

$B (k, 1) = \lfloor \log n \rfloor + 1$

A figura (II.6) ilustra o sistema escolhido, em escala logarítmica. Não foi representada a partição 0. A região assinalada corresponde a bloco (2, 2).



SPM para Compressão e União por Tamanho

Figura II.6

Vamos estimar valores para N_{pq} e R_{pq} levando em conta o sistema escolhido:

a. $R_{p0} = 2$, $1 \leq p < k$, pois

$$\begin{aligned} \text{bloco}(p, 0) &= \{ x \in \mathbb{N} \mid B(p, 0) \leq x < B(p, 1) \} \\ &= \{ x \in \mathbb{N} \mid 0 \leq x < A(p, 1) \} \\ &= \{ x \in \mathbb{N} \mid 0 \leq x < A(p-1, 2) \} \\ &= \text{bloco}(p-1, 0) \cup \text{bloco}(p-1, 1). \end{aligned}$$

b. $R_{pq} \leq A(p, q)$, $1 \leq p < k$, $1 \leq q < L_p$, pois

$$\begin{aligned} \text{bloco}(p, q) &= \{ x \in \mathbb{N} \mid B(p, q) \leq x < B(p, q+1) \} \\ &= \{ x \in \mathbb{N} \mid A(p, q) \leq x < A(p, q+1) \} \\ &\subseteq \{ x \in \mathbb{N} \mid 0 \leq x < A(p-1, A(p, q)) \} \end{aligned}$$

c. $R_{k0} \leq \lfloor (m+n)/n \rfloor$, pois

$R_{k0} = L_{k-1} = \min \{ q \mid A(\alpha(m+n, n), q) > \log n \}$

Pela definição de α, temos:

$\alpha(m+n, n) = \min \{ p \geq 1 \mid A(p, \lfloor (m+n)/n \rfloor) > \log n \}$

Logo, $R_{k0} \leq \lfloor (m+n)/n \rfloor$.

d. $N_{p0} \leq n$, obviamente.

e. $N_{pq} \leq n / 2^{A(p, q-1)}$, $1 \leq p < k$, $1 \leq q < L_p$.

Pelo lema (II.4), existem no máximo $n / 2^u$ vértices de posto u. Portanto

$$N_{pq} \leq \sum_{u=B(p,q)}^{B(p,q+1)-1} n / 2^u$$

ou

$$N_{pq} \leq \sum_{u=A(p,q)}^{A(p,q+1)-1} n / 2^u$$

ou

$$N_{pq} \leq \sum_{u \geq A(p,q)} n / 2^u$$

ou

$$N_{pq} \leq n / 2^{(A(p,q)-1)}.$$

O duplo somatório S pode ser separado em três partes $S = S_1 + S_2 + S_3$, onde:

$$S_1 = \sum_{q=0}^{L_k-1} N_{kq} \cdot (R_{kq} - 1) =$$

$$= N_{k0} \cdot (R_{k0} - 1), \text{ pois } L_k = 1.$$

Ou

$$S_1 \leq n \cdot (\lfloor (m+n)/n \rfloor - 1) \leq m + n.$$

$$S_2 = \sum_{p=1}^{k-1} N_{p0} \cdot (R_{p0} - 1) = \sum_{p=1}^{k-1} N_{p0}$$

ou

$$S_2 \leq n \cdot (k - 1)$$

$$S_3 = \sum_{p=1}^{k-1} \sum_{q=1}^{L_p-1} N_{pq} \cdot (R_{pq} - 1)$$

$$\leq \sum_{p=1}^{k-1} \sum_{q=1}^{L_p-1} (n \cdot A(p, q)) / 2^{(A(p,q)-1)}$$

ou

$$S_3 \leq \sum_{p=1}^{k-1} \sum_{x \geq A(p,1)} (n \cdot x) / 2^{x-1}$$

Ou

$$S_3 \leq \sum_{p=1}^{k-1} (n \cdot (A(p, 1) + 1)) / 2^{(A(p,1)-2)}$$

Finalmente

$$S_3 \leq n \cdot \sum_{x \geq 2} (x + 1) / 2^{x-2} = 8n$$

Portanto, $ND \leq S \leq S1 + S2 + S3$. Ou

$$ND \leq m + n + n \cdot (k - 1) + 8n \leq m + 8n + nk$$

O tempo total, como foi afirmado inicialmente, é proporcional a NC + ND, ou $2m + 9n + (m + n).k - 1$. Logo a sequência é executada em tempo

$$O((m + n).\ k) = O((m + n) \cdot \propto(m + n,\ n)).$$

Cacolácio II.2. Se $m \geq n$, a sequência de até $n - 1$ UNIONs e m FINDs é executada em tempo $O(m.\propto(m,\ n))$ usando-se compressão de caminhos e união por tamanho.

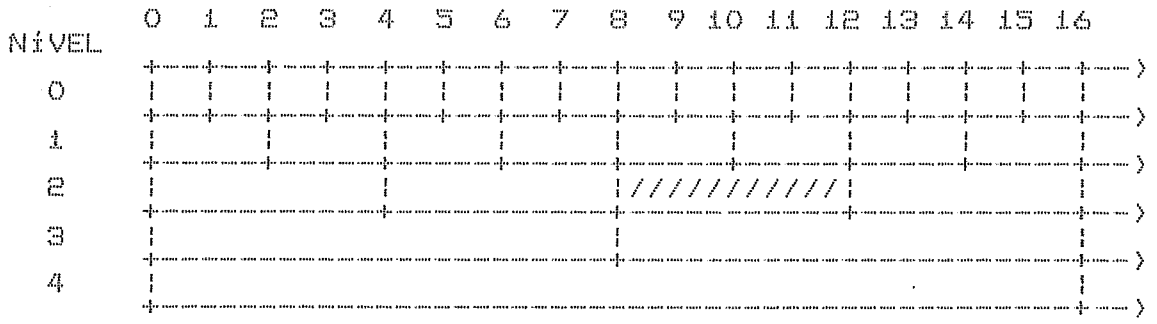
Prova. Basta ver que, se $m \geq n$, $\propto(m + n,\ n) < \propto(m,\ n)$.

Teorema II.2. Usando compressão de caminhos e união sem a regra de tamanho (*naive link*), a sequência de até $n - 1$ UNIONs e $m \geq n$ FINDs sobre um universo de n elementos é executada em tempo $O(m \cdot \log_{\lfloor 1 + m/n \rfloor} n)$.

Prova. Seja $X = \lfloor 1 + m / n \rfloor$ e o seguinte SPM:

$$\begin{aligned} k &= \lceil \log_X n \rceil \\ L_p &= \lceil n / X^p \rceil \\ B(p, q) &= q \cdot X^p \\ U &= n \end{aligned}$$

A figura (II.7) ilustra o sistema escolhido, quando $X = 2$. A porção hachurada corresponde a bloco (2, 2).



SPM para Compressão e *Naive Link*, quando X = 2

Figura II.7

Neste sistema de partições, temos $R_{pq} = X$. O duplo somatório S vale então:

$$S = \sum_{p=1}^{k} \sum_{q=0}^{L_p - 1} N_{pq} \cdot (R_{pq} - 1)$$

$$\leq (X - 1) \cdot \sum_{p=1}^{k} \sum_{q=0}^{L_p - 1} N_{pq}$$

$$\leq \lfloor (n + m) / n \rfloor \cdot \sum_{p=1}^{k} n$$

$$\leq \lfloor (n + m) / n \rfloor \cdot kn \leq k.(n + m)$$

O tempo total é proporcional a NC + ND, ou seja, $n - 1 + k.(m + 1) + k.(n + m)$. Logo, a sequência é executada em tempo $O(m \cdot \log_{\lfloor 1 + m/n \rfloor} n)$.

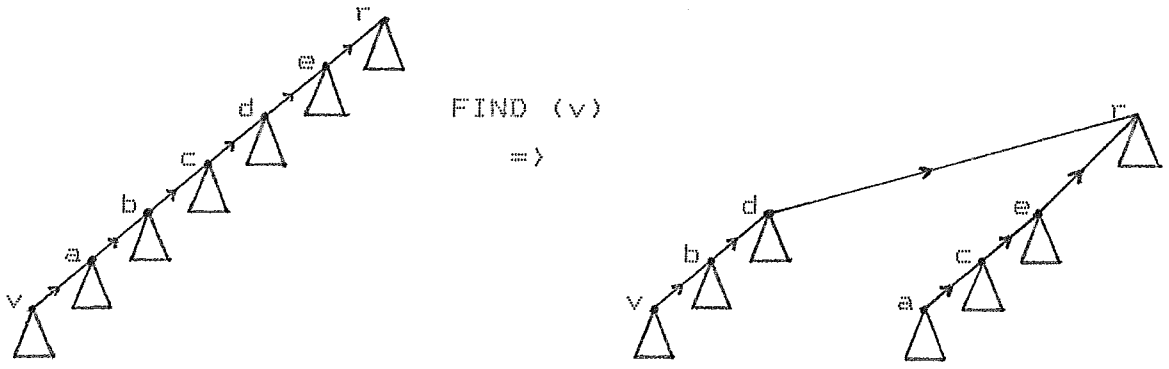
Cabe-nos aqui tecer alguns comentários sobre o limite inferior para o problema de executar até n - 1 UNIONs e m FINDs. Baseado em um modelo computacional denominado *pointer machine*, TARJAN [12] mostrou que, dados m e n, existe sempre

uma sequência de UNIONs e FINDs que leva tempo O ((m + n) . α (m + n, n)) para ser executada. Desta forma, os *algoritmos alpha* desenvolvidos para implementar UNION e FIND são ótimos para este modelo computacional, genérico o suficiente para abranger uma grande variedade de arquiteturas conhecidas.

II.8. Outras Formas de Compactação.

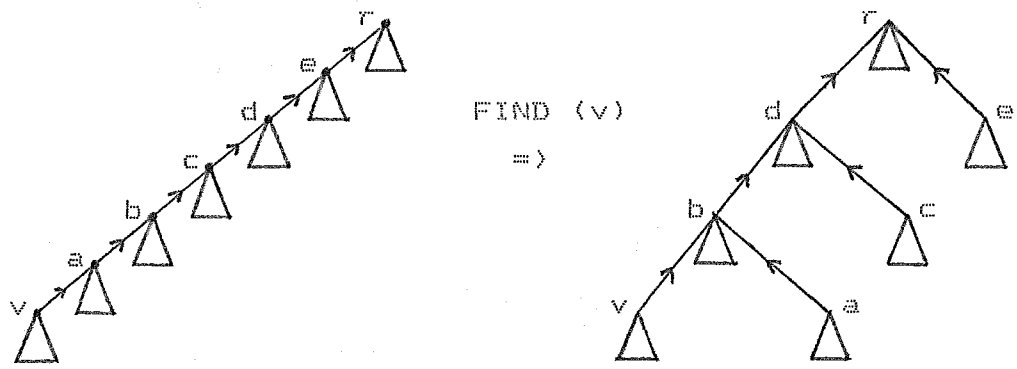
Além da compressão, que possui o inconveniente de percorrer duas vezes o caminho (algoritmo (II.6)), VAN LEEUWEN & TARJAN analisam em [15] diversas outras formas de compactação, duas dentre as quais resultam em algoritmos de complexidades idênticas à analisada na seção (II.7), evitando o duplo percurso até a raiz da árvore.

A primeira delas, denominada *splitting*, consiste em tornar cada vértice filho de seu avô, com exceção do penúltimo, dividindo o caminho em dois, conforme ilustra a figura (II.8).



Splitting de um Caminho
Figura II.8

A outra técnica denomina-se *halving* e está ilustrada na figura (II.9).



Halving de um Caminho
Figura II.9

No algoritmo (II.8) implementamos a operação FIND utilizando *splitting* e *halving*.

Algoritmo II.8: FIND com *Splitting* e *Halving*.

```
func SPLIT_FIND (x : inteiro) : inteiro;
var
    y, z : inteiros;
início
    y := pai [x];
    Se y = 0 então
        Retorne (nome [x]);
    z := pai [y];
    Enquanto z ≠ 0 faça
        pai [x] := z;
        x := y;
        y := z;
        z := pai [y]
    fim;
    Retorne (nome [y]);
fim;
```

```
func HALF_FIND (x : inteiro) : inteiro;
var
    y, z : inteiros;
início
    y := pai [x];
    Se y = 0 então
        Retorne (nome [x]);
    z := pai [y];
    Enquanto z ≠ 0 faça
        pai [x] := z;
        x := z;
        y := pai [x];
        Se y = 0 então
            Retorne (nome [x]);
        z := pai [y]
    fim;
    Retorne (nome [y]);
fim;
```

II.9. União por Posto.

Uma técnica equivalente à união por tamanho proposta por VAN LEEUWEN & TARJAN [15] denomina-se *União por Posto*. Ao invés de mantermos com cada vértice v a informação tam [v], mantemos posto [v]. Inicialmente, posto [v] = 0, para todo vértice v. Ao combinar duas árvores de raizes v e w, comparamos posto [v] e posto [w]: se posto [v]) posto [w], acrescentamos a aresta (w, v); se posto [v] = posto [w], acrescentamos a aresta (w, v) e incrementamos de uma unidade posto [v]; finalmente, se posto [v] (posto [w], acrescentamos a aresta (v, w). O algoritmo (II.9) ilustra esta estratégia.

Algoritmo II.9: União por Posto.

```
var
    pai, raiz,
    nome, posto : arranjos [1..n] de inteiros;

proc UNION_BY_RANK (A, B, C : inteiros);
var
    r_a, r_b, r_c : inteiros;
início
    r_a := raiz [A];
    r_b := raiz [B];
    r_c := raiz [C];
    Se r_a = 0 ou r_b = 0 então
        Erro (A, " ou", B, " não são raizes");
        Retorne
    fim;
    Se r_c ≠ r_a e r_c ≠ r_b e r_c ≠ 0 então
        Erro ("Conflito de Nomes na Coleção");
        Retorne
    fim;
    Se r_a = rb então
        Retorne;
    raiz [A] := raiz [B] := 0;
    Se posto [r_a] < posto [r_b] então
        Troque r_a e r_b
    Caso contrário
        Se posto [r_a] = posto [r_b] então
            posto [r_a] := posto [r_a] + 1
    fim;
    pai [r_b] := r_a;
    raiz [C] := r_a;
    nome [r_a] := C
fim;
```

Observe que a hipótese de crescimento estrito dos postos nos caminhos é mantida: quando ocorre a igualdade no momento da união, o posto do sucessor é incrementado. Além disso,

demonstram-se analogamente, para união por posto, os mesmos resultados obtidos na seção (II.4) para união por tamanho, sendo ambas as formas equivalentes.

Comentam VAN LEEUWEN & TARJAN [15] que a união por posto é preferível à união por tamanho, pois, embora a complexidade não seja afetada, a união por posto requer menos alterações na informação posto [v] e menos espaço para armazenamento, já que os postos variam de 0 a log n e só necessitamos de log log n bites para a representação, ao passo que tam [v] varia de 1 a n, exigindo log n bites.

CAPÍTULO III

UMA ALTERNATIVA LINEAR

Neste capítulo, analisamos um caso particular do problema de União de Conjuntos Disjuntos, no qual a árvore final obtida pela sequência de operações UNION é conhecida *a priori*. Assim sendo, é possível obter um algoritmo linear para resolver o problema, muito embora este resultado tenha destaque essencialmente em seu aspecto teórico.

III.1. O Caso Particular da União de Conjuntos Disjuntos.

Seja o Problema da União de Conjuntos Disjuntos, discutido na seção (II.6). Conforme vimos, a sequência de n - 1 operações UNION transforma a floresta de n árvores triviais em uma floresta constituída por uma única árvore de n vértices, que denominamos *árvore de referência*. Em algumas aplicações, esta árvore é conhecida antes mesmo de a sequência de UNIONs e FINDs ser executada e, de posse desta preciosa informação, podemos obter um algoritmo que execute a sequência em tempo linear.

Para tal, chamemos de T a árvore de referência com n vértices e, para um determinado vértice v ∈ T, seja pai (v) o vértice pai de v em T; no caso especial da raiz de T, supomos pai (raiz) = NULL. O algoritmo desenvolvido neste capítulo implementa duas operações denominadas T-UNION e T-FIND, definidas assim:

```
T-UNION (v)        u := FIND (v); w := FIND (pai (v));
                   UNION (w, u, w);
T-FIND  (x)        FIND  (x)
```

O algoritmo que apresentaremos encontra-se em [3]. Sua idéia

central consiste em combinar os algoritmos anteriormente apresentados com consultas a pequenas tabelas construídas em uma etapa inicial; mais precisamente, trata-se de associar, por exemplo, a técnica de união por tamanho sem compressão de caminhos sobre um universo com menos do que n elementos a consultas a tabelas, obtendo artesanalmente uma complexidade linear.

III.2. Microconjuntos e a Operação T-UNION.

Vamos dividir o conjunto de vértices da árvore T em pequenos subconjuntos não-vazios e disjuntos denominados *microconjuntos*. Este particionamento deve obedecer às três condições seguintes:

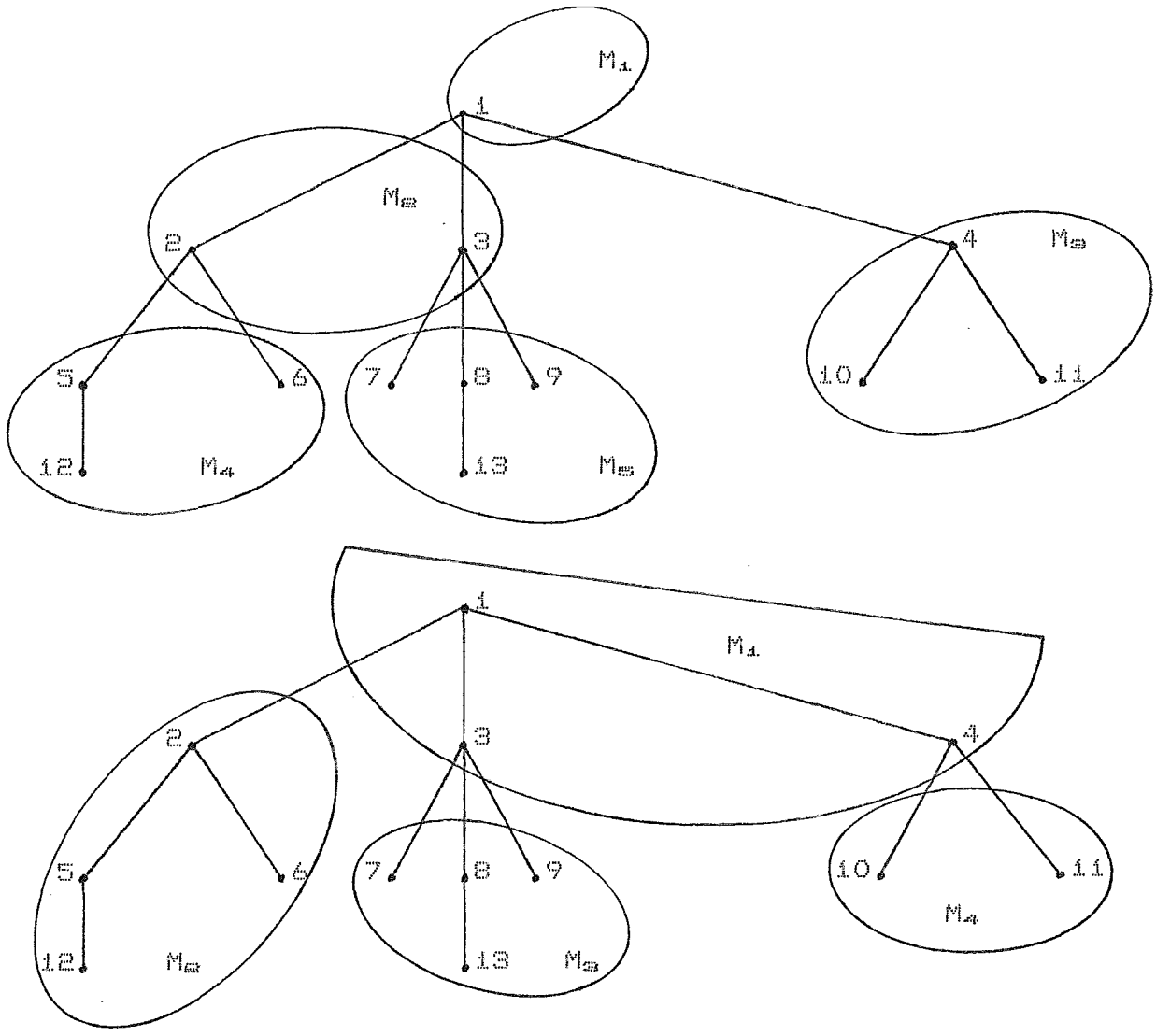
(i) Todo microconjunto tem, no máximo, B - 1 elementos, onde B será fixado oportunamente;

(ii) Há, no total, O (n / B) microconjuntos;

(iii) Para todo microconjunto M, existe um vértice x não pertencente a M, denominado *raiz* de M, de forma que, para todo vértice v $\in$ M, tem-se pai (v) $\in$ M $\cup$ { x }. Em outras palavras, um microconjunto M induz na árvore de referência T uma floresta e o conjunto M $\cup$ { x } induz em T uma sub-árvore, sendo x pai dos vértices que são raizes das sub-árvores que constituem a floresta induzida por M. Em particular, o microconjunto que contém a raiz de T tem como raiz NULL.

Ressaltamos desde já que o particionamento de T em microconjuntos é realizado em uma etapa inicial do algoritmo e permanece fixo com o decorrer da execução da seqüência de T-UNIONs e T-FINDs, nada tendo a ver com os conjuntos que vão sendo formados por esta seqüência. Ademais, o

particionamento de T segundo os critérios acima não é único para um dado B, ao passo que a sequência de partições decorrentes das operações T-UNION é única. A figura (III.1) mostra duas partições distintas de uma árvore T com 13 nós para B = 5.



Duas Partições para uma Árvore

Figura III.1

Os microconjuntos em que T é particionada são numerados a partir de 1 e, dentro de cada microconjunto, numeramos os vértices também a partir de 1 segundo um percurso em pré-ordem da floresta por ele induzida. Como, por definição, um vértice v não pode pertencer a dois microconjuntos simultaneamente, podemos individualizá-lo biunivocamente pelo par ordenado

(micro (v), num (v))

onde micro (v) é o número do microconjunto que contém { v } e num (v) é o número em pré-ordem de v em micro (v). A figura (III.2) mostra esta correspondência para o primeiro particionamento da figura (III.1); m é a numeração dos microconjuntos gerados.

m	raiz (m)
1	NULL
2	1
3	1
4	2
5	3

v	micro (v)	num (v)
1	1	1
2	2	1
3	2	2
4	3	1
5	4	1
6	4	3
7	5	1
8	5	2
9	5	4
10	3	2
11	3	3
12	4	2
13	5	3

Tabelas Correspondentes a um Particionamento
Figura III.2

Para implementar a operação T-UNION, definimos para cada vértice v ∈ T:

$$\text{marca}(v) = \begin{cases} 0 & \text{se o vértice } v \text{ é nome de um conjunto} \\ 1 & \text{caso contrário} \end{cases}$$

Inicialmente, como todo vértice v ∈ T constitui um conjunto unitário (árvore trivial), temos *marca* (v) = 0. Ao realizarmos a operação T-UNION (v), v deixa de ser raiz de uma árvore e passa a fazer parte da árvore na qual pai (v) está situado. Logo,

T-UNION (v) => marca (v) := 1

Visando a implementação, tecemos as seguintes considerações:

a. Podemos armazenar as marcas relativas aos vértices de um mesmo microconjunto em um número inteiro, que denominaremos *máscara* do microconjunto, utilizando um bite para cada vértice. Estas máscaras estão armazenadas no vetor *máscara*.

b. Para v ∈ T, os vetores *micro* e *num* armazenam o número do microconjunto e a posição de v dentro do microconjunto.

c. Inversamente, a matriz *vértice* é construída de forma que vértice [m, n] = v <=> micro [v] = m e num [v] = n.

Com base nestas observações, a implementação de T-UNION é mostrada no algoritmo (III.1). O operador "|" calcula um "ou bite a bite" dos operandos, ao passo que a expressão "a << b" significa a * 2^b.

Algoritmo III.1: Implementação de T-UNION.

```
proc T-UNION (v : inteiro);
var
    m, n : inteiros;
início
    m := micro [v];
    n := num [v] - 1;
    máscara [m] := máscara [m] | (1 << n)
fim;
```

Assim expressa, a operação T-UNION leva tempo O (1) para ser executada.

III.3. A Operação T-FIND.

A operação T-FIND (x) deve retornar como resultado um nó v, ancestral mais próximo de x (possivelmente ele próprio), tal que marca (v) = 0, ou seja, v é raiz (nome) do conjunto que contém { x }. Este cálculo alterna duas etapas:

a. Busca dentro de um Microconjunto.
   Nesta fase buscamos o ancestral mais próximo de x que esteja marcado e pertença ao mesmo microconjunto ao qual x pertence, ou seja, micro (x). Se existir tal ancestral, ele será o resultado de T-FIND (x). Caso ele não exista, localizamos a raiz de micro (x) que, por construção, situa-se em outro microconjunto e procedemos ao passo b.

b. Busca entre Microconjuntos.
   Para esta etapa, mantemos uma coleção de conjuntos disjuntos constituídos pelas raizes dos microconjuntos. Estes conjuntos são denominados *macroconjuntos* e são manipulados pelas operações MACRO-UNION e MACRO-FIND, semelhantes a UNION e FIND do Capítulo II:

   MACRO-UNION (a, b): Promove a união dos conjuntos que contêm { a } e { b }, gerando um novo conjunto cujo nome é o mesmo do antigo conjunto que continha { a }.

   MACRO-FIND (c): Devolve o nome do conjunto que contém { c }.

Observe que a operação MACRO-UNION difere da operação UNION no seguinte aspecto: MACRO-UNION tem como operandos elementos de conjuntos, ao passo que UNION tinha como

operandos *nomes* de conjuntos. Isto significa que, antes de procedermos à união propriamente dita dos conjuntos, devemos localizar suas raizes (nomes), através de um MACRO-FIND, ou seja

MACRO-UNION (a, b) => UNION (FIND (a), FIND (b), FIND (a))
MACRO-FIND (c) => FIND (c)

A implementação destas operações será discutida na seção (III.9) e, conforme será visto, uma sequência de até w - 1 operações MACRO-UNION e z operações MACRO-FIND sobre um universo de w elementos pode ser executada em tempo O (z + w log w), usando a idéia de união por tamanho sem compressão de caminhos.

Supondo a existência do procedimento *microfind (x)*, que realiza as ações descritas no passo a, T-FIND (x) é implementada no algoritmo (III.2).

Algoritmo III.2: Implementação de T-FIND.

```
    func T-FIND (x : inteiro) : inteiro;
    var
        y : inteiro;
    início
(1)     y := microfind (x);
(2)     Se micro (x) ≠ micro (y) então
(3)         x := MACRO-FIND (y);
(4)         y := microfind (x);
(5)         Enquanto micro (x) ≠ micro (y) faça
(6)             MACRO-UNION (y, x);
(7)             x := MACRO-FIND (x);
(8)             y := microfind (x)
            fim
        fim;
        Retorne (y)
    fim;
```

O teorema (III.1) que enunciaremos a seguir estima o tempo para executar m T-UNIONs e T-FINDs sob certas restrições.

Teorema III.1. Suponha B = Ω(log n) e que microfind (x) leve tempo O (1). Então uma sequência de n - 1 T-UNIONs e m T-FINDs é executada em tempo O (m + n) usando-se as implementações mostradas (algoritmos (III.1) e (III.2)).

Prova.

As operações T-UNION perfazem, ao todo, O (n) em tempo. Quanto aos T-FINDs, observemos que, se MACRO-UNION é executado, então necessariamente x e y estão em macroconjuntos diferentes, graças ao teste micro (x) ≠ micro (y) realizado sempre após um MACRO-FIND. Como existem O (n / B) macroconjuntos no máximo, MACRO-UNION é executado, no máximo, O (n / B) vezes. A tabela a seguir demonstra a quantidade máxima de vezes que cada operação é executada.

OPERAÇÃO	Nº MÁXIMO DE VEZES
MACRO-UNION	O (n / B)
MACRO-FIND (total) . na linha (3) . na linha (7)	m + O (n / B) m O (n / B)
microfind (total) . na linha (1) . na linha (4) . na linha (8)	2m + O (n / B) m m O (n / B)

Uma sequência de m + O (n / B) MACRO-FINDs e O (n / B) MACRO-UNIONs sobre um universo de O (n / B) elementos é

executada em tempo

$$O(m + O(n / B) + O(n / B) \cdot \log O(n / B)).$$

Adicionando-se a isso o tempo de execução dos microfinds, temos o tempo total também dado por

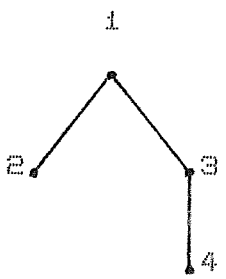
$$O(m + O(n / B) + O(n / B) \cdot \log O(n / B))$$

que é $O(m + n)$ se $B = \Omega(\log n)$.

III.4. Determinação de MICROFIND.

Conforme pressuposto no teorema (III.1), *microfind (x)* deve levar tempo O (1) para que a sequência de T-UNIONs e T-FINDs seja executada em tempo linear. Para tanto, associamos a cada microconjunto M uma tabela denominada *ancestrais*, que é uma matriz de 2^b x b elementos, onde b é o número de vértices de M. Nesta tabela encontram-se calculados, para cada vértice $v \in M$, o ancestral mais próximo de v que esteja marcado e também pertença a M, levando-se em conta todas as variações possíveis da máscara (são, ao todo, 2^b variações).

Seja, por exemplo, o microconjunto da figura (III.3) de 4 vértices, numerados em pré-ordem.



Um Microconjunto de 4 Vértices

Figura III.3

A tabela *ancestrais* para este microconjunto é mostrada na figura (III.4).

	MÁSCARA	Nó			
	1 2 3 4	1	2	3	4
	0 0 0 0	1	2	3	4
	0 0 0 1	1	2	3	3
	0 0 1 0	1	2	1	4
(*)	0 0 1 1	1	2	1	1
	0 1 0 0	1	1	3	4
	0 1 0 1	1	1	3	3
	0 1 1 0	1	1	1	4
	0 1 1 1	1	1	1	1
	1 0 0 0	0	2	3	4
	1 0 0 1	0	2	3	3
	1 0 1 0	0	2	0	4
	1 0 1 1	0	2	0	0
	1 1 0 0	0	0	3	4
	1 1 0 1	0	0	3	3
	1 1 1 0	0	0	0	4
	1 1 1 1	0	0	0	0

Tabela *ancestrais* para o Microconjunto da figura (III.2)

Figura III.4

Analisemos a linha assinalada com (*) na tabela da figura (III.4). A máscara correspondente é "0 0 1 1", o que significa que os vértices 3 e 4 não estão marcados. Logo, o ancestral mais próximo dos vértices 3 e 4 que está marcado é 1, conforme mostram as entradas respectivas.

As posições da tabela preenchidas com zero indicam que nenhum ancestral (dentro do microconjunto) do referido vértice está marcado e que microfind deverá, neste caso, retornar como resultado a raiz do microconjunto em questão.

Assim sendo, a implementação de *microfind* é feita no algoritmo (III.3). O vetor *raiz* armazena os vértices que são raizes dos microconjuntos.

Algoritmo III.3: Implementação de *microfind*.

```
func microfind (x : inteiro) : inteiro;
var
    k, m : inteiros;
início
    m := micro [x];
    a := ancestrais [m];
    k := a [máscara [m], num [x]];
    Se k > 0 então
        Retorne (vértice [m, k])
    Caso contrário
        Retorne (raiz [m])
fim;
```

III.5. O Cálculo da Tabela Ancestrais.

O cálculo da tabela *ancestrais* para um microconjunto M é trivial. Aproveitando o percurso em pré-ordem que deverá ser feito para numerar os vértices de M, calculamos para cada vértice $v \in M$:

$$p\,[num\,(v)] = \begin{cases} num\,[pai\,(v)], & \text{se } pai\,(v) \in M \\ 0, & \text{caso contrário} \end{cases}$$

Em seguida, fazemos o cálculo da tabela. Uma variável i controla a variação da máscara, indo de 0 a 2^m-1. Cada valor de i corresponde a uma diferente distribuição de marcas para os vértices de M. A variável j vai de 1 a b, correspondendo

aos vértices de M. Se j for um vértice marcado, ele será seu próprio ancestral e fazemos ancestrais [i, j] := j; se j não estiver marcado, seguimos os ponteiros começando em p [j] até encontrar um ancestral marcado k e fazemos ancestrais [i, j] := k. Para agilizar esta busca, é introduzido o vetor *aux*, que guarda o ancestral mais próximo marcado e é atualizado a cada passo. A função bite (a, b) devolve o valor do b-ésimo bite de a, se b > 0, ou 0 se b ≤ 0. A implementação é mostrada no algoritmo (III.4).

Algoritmo III.4: Cálculo da Tabela Ancestrais.

```
proc calcula_ancestrais (p, b);
var
    i, j, k : inteiros;
    aux : arranjo [1..b] de inteiros;
início
    Para i := 0 até 2^b-1 faça
        Para j := 1 até b faça
            aux [j] := p [j];
            Se bite (i, j) = 0 então
                ancestrais [i, j] := j
            Caso contrário
                k := aux [j];
                Se bite (i, k) = 1 então
                    k := aux [k];
                    aux [j] := k
                fim;
                ancestrais [i, j] := k
            fim
        fim
    fim
fim;
```

Examinando o algoritmo (III.4), constatamos que o cálculo da tabela ancestrais para um microconjunto de b vértices leva tempo O (b . 2^b).

Um fato notável acerca da tabela ancestrais é que ela depende unicamente da disposição dos vértices do microconjunto (formato da floresta). Com isto, se o particionamento de T gerar microconjuntos com organizações idênticas, a tabela ancestrais só precisa ser calculada uma vez para cada organização que decorra do particionamento. Como a organização de T pode ser qualquer, não podemos de antemão prever quais serão as organizações dos microconjuntos para estimarmos precisamente a complexidade do cálculo de todas as tabelas ancestrais referentes a T. Ao invés, vamos supor o pior caso: o particionamento de T em face do parâmetro B gera todas as organizações possíveis; calculando a complexidade para este caso extremo, temos um limite superior para qualquer situação que ocorra na prática.

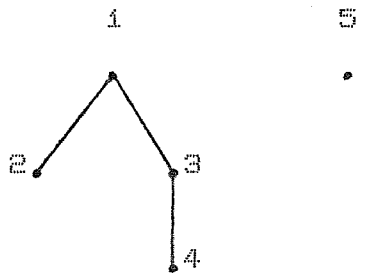
Definição III.1. Seja um microconjunto M com b vértices numerados em pré-ordem. Denominamos *característica* de M ao número inteiro cuja representação binária é a seguinte: cada vértice é representado por uma cadeia de bites começando com 1 seguido de tantos zeros quantos forem seus filhos; no final, as cadeias são concatenadas, formando um número inteiro.

$$1\ 0\ \ldots\ 0\ 1\ 0\ \ldots\ 0\ \ldots\ 1\ 0\ \ldots\ 0\ 1$$
$$f_1 \qquad\qquad f_2 \qquad\qquad f_{b-1}$$

onde f_j (quantidade de zeros após o 1) é o número de filhos do vértice numerado j.

Observe que a característica é sempre um número ímpar, uma vez que o vértice número b não pode ter filhos.

Por exemplo, a figura (III.5) mostra um microconjunto de 5 vértices e sua característica.



v	filhos
1	2
2	0
3	1
4	0
5	0

Característica = $100110111_{(2)}$ = 155

Um Microconjunto e sua Característica
Figura III.5

Pode-se notar que a característica determina biunivocamente a organização do microconjunto. Em outras palavras, dado um microconjunto temos uma única característica associada a ele (calculada de acordo com a definição (III.1)) e, reciprocamente, dado um número inteiro, ou não existe microconjunto cuja característica seja igual a ele ou existe um único.

Lema III.1. Seja um microconjunto M com b vértices. Então sua característica é um número inteiro entre

$$2^b - 1 \quad e \quad (2^{2b} - 1) / 3$$

Prova. O limite inferior é imediato e corresponde à organização em que todos os vértices da floresta têm 0 filhos (pontos isolados).

O limite superior corresponde à configuração que resulta na maior representação binária possível, na qual cada vértice tem exatamente um filho (exceto o último). Esta configuração, em binário, vale $10_{(2)}10_{(2)}\ldots10_{(2)}1_{(2)}$, sendo b - 1 grupos "10". Logo,

$$1010\ldots101_{(2)} = 1 + \sum_{j=1}^{b-1} 2^{2j} = 1 + (2^{2b} - 4) / 3$$

$$= (2^{2b} - 1) / 3$$

Teorema III.2 O tempo para calcular as tabelas ancestrais correspondentes a todas as organizações possíveis de microconjuntos com menos de B vértices é $O(B \cdot 2^{(B-1)})$.

Prova. Pela condição *(i)* de particionamento, os microconjuntos têm de 1 a B − 1 vértices. O tempo exato para calcular todas as tabelas é proporcional ao somatório:

$$S = \sum_{j=1}^{B-1} t_j . j . 2^j$$

onde t_j é o número de organizações distintas que podemos formar com j vértices. Como a determinação de t_j não nos parece trivial, limitaremos S superiormente fazendo as seguintes aproximações:

a. Supondo exageradamente que todos os microconjuntos terão B − 1 vértices, temos:

$$S < (B - 1) . 2^{(B - 1)} . \sum_{j=1}^{B-1} t_j$$

b. O somatório restante representa o número de organizações possíveis para 1, 2, ..., B − 1 vértices. De acordo com o lema (III.1), este número pode variar entre 1 e $(2^{2B-2} - 1) / 3$ e, pela definição, deve ser ímpar. Como existem $(2^{(2B-3)} + 1) / 3$ números ímpares neste intervalo

$$S < (B - 1) . 2^{(B - 1)} . 2^{(2B - 3)}$$

ou

$$S = O(B \cdot 2^{(m-4)})$$

III.6. Divisão da Árvore de Referência em Microconjuntos.

A divisão de T em microconjuntos que satisfaçam às três condições apresentadas na seção (III.2) é baseada em um percurso de T em pós-ordem. Associamos a cada vértice $v \in T$ as seguintes informações:

a. *resto (v)*: Lista formada pelos filhos de v que ainda não foram incluídos em nenhum microconjunto;

b. *nfilhos (v)*: Número de elementos em resto (v);

c. *cont (v)*: O número de descendentes (não somente filhos!) de v ainda não incluídos em nenhum microconjunto (inclusive o próprio v).

Sempre que um microconjunto é formado tendo raiz em v, os vértices que o constituem são todos os descendentes de v ainda não incluídos em nenhum microconjunto. Matematicamente, estes vértices formam o conjunto

$$D = \bigcup_{j \geq 1} D_j$$

onde

$$D_1 = \{ x \mid x \in \text{resto}(v) \}$$

$$D_{j+1} = \bigcup_{w \in D_j} \{ x \mid x \in \text{resto}(w) \}, \text{ se } j > 0$$

O conjunto D possui cont (v) - 1 elementos.

O algoritmo (III.5) mostra o procedimento *particiona*. Este procedimento chama *pós-ordem* para percorrer a árvore T em pós-ordem formando os microconjuntos e, ao final, constrói um último microconjunto, que contém a raiz de T. A função *percorre* percorre o microconjunto M sendo construído em pré-ordem, determinando para cada v ∈ M, micro [v], num [v], vértice [micro [v], num [v]] = v e p [num [v]] (este último é usado no cálculo da tabela ancestrais, mostrado no algoritmo (III.4)).

Algoritmo III.5: Divisão da Árvore em Microconjuntos.

```
        proc pós_ordem (v : inteiro);
        var
            w : inteiro;
        início
            w := primeiro filho de v;
            Enquanto w ≠ NULL faça
                pós-ordem (w);
                w := próximo filho de v
            fim;

            cont [v] := 1;
            nfilhos [v] := 0;
            resto [v] := lista_vazia;
            w := primeiro filho de v;

    volta:
(*)         Enquanto (cont [v] < (B+1)/2) e (w ≠ NULL) faça
                cont [v] := cont [v] + cont [w];
                Acrescente w ao final da lista resto [v];
                nfilhos [v] := nfilhos [v] + 1;
                w := próximo filho de v
            fim;

(**)        Se cont [v] ≥ (B+1)/2 então
                nmicro := nmicro + 1;
                raiz [nmicro] := v;
```

```
                pnum := -1;
                carac := 0;
                percorre (v);
                máscara [nmicro] := 0;

(***)            Se não_calculado [(carac + 1) / 2] então
                    calcula_ancestrais (p, pnum);
                    não_calculado [(carac + 1) / 2] := falso
                fim;

                cont [v] := 1;
                nfilhos [v] := 0;
                resto [v] := lista_vazia;
                Vá para volta
            fim
        fim;


        func percorre (v : inteiro) : inteiro;
        var
            w : inteiro;
        início
            pnum := pnum + 1;
            Se pnum > 0 então
                num [v] := pnum;
                micro [v] := nmicro;
                vértice [nmicro, pnum] := v;
                w := nfilhos [v];
                carac := (carac << (w + 1)) | (1 << w)
            fim;

            w := primeiro elemento de resto [v];
            Enquanto w ≠ NULL faça
                p [percorre (w)] := pnum;
                w := próximo elemento de resto [v]
            fim;
            Retorne (pnum)
        fim;
```

```
        proc particiona (T);
        var
            i : inteiro;
        início
            Para i := 1 até (2^(e-a) + 1) / 3 faça
                não_calculado [i] := verdadeiro;
            nmicro := 0;
            pós_ordem (raiz de T);

            nmicro := nmicro + 1;
            carac := pnum := 0;
            percorre (raiz de T);
            máscara [nmicro] := 0;
            raiz [nmicro] := NULL;

(***)        Se não_calculado [(carac + 1) / 2] então
                calcula_ancestrais (p, num);
                não_calculado [(carac + 1) / 2] := falso
            fim
        fim;
```

Observações sobre o Algoritmo de Particionamento:

a. Ao atingir a linha (*) durante o processamento de um vértice v ∈ T, todos os vértices w filhos de v já terão sido processados e podemos dispor dos valores cont [w] e resto [w].

b. A chamada a calcula_ancestrais deve ser feita somente uma vez para cada organização, a fim de que o limite estabelecido no teorema (III.2) permaneça válido. Por esta razão, foram incluídos os testes (***) para verificar se a organização (em função da característica) já ocorreu antes. O vetor *não_calculado* requer tantos bites quantos forem os números ímpares entre 1 e $(2^{e-a} - 1) / 3$, ou seja, $(2^{e-a} + 1) / 3$ bites.

Teorema III.3. O algoritmo (III.5) está correto, ou seja, os microconjuntos por ele gerados satisfazem às condições *(i)*, *(ii)* e *(iii)* da seção (III.2).

Prova.

(i) Com efeito, o laço assinalado com (*) no algoritmo é abandonado quando cont [v] $\geq$ (B+1)/2 ou quando se esgotam os filhos de v. Ao fim deste laço, o valor de cont [v] será no máximo B, pois cada vértice w filho de v contribui com no máximo (B+1)/2 - 1 = (B-1)/2 vértices e a soma cont [v] + cont [w] não excederá B ao fim do laço.

(ii) Cada conjunto tem pelo menos (B+1)/2 - 1 vértices, de acordo com o teste (**). Logo, o número máximo de microconjuntos será

$$2n / (B-1) + 1$$

incluindo o último microconjunto formado (o que contém a raiz de T).

(iii) As raizes das árvores da floresta são vértices filhos de um mesmo v, que será raiz do microconjunto.

Teorema III.4. O procedimento *particiona* executa em tempo O (n).

Prova. Basta observar que o percurso em pós-ordem leva tempo O (n) e o laço (*) é executado uma vez para cada filho w de cada vértice v $\in$ T (O (n)). Quanto à formação dos microconjuntos, para construir um deles (sem considerar a execução de calcula_ancestrais) levamos tempo proporcional ao número de vértices que o constituem, perfazendo O (n) ao total. Reunindo todas as chamadas a calcula_ancestrais com a ressalva feita na Observação b, se o tempo para calculá-las

(uma vez apenas para cada organização) for também O (n), o algoritmo será linear.

III.7. Estimativa para a Cardinalidade Máxima dos Microconjuntos.

Para completar o algoritmo, falta-nos fazer uma estimativa para o parâmetro B, deixado em aberto desde a seção (III.2). Este parâmetro deve satisfazer às seguintes restrições:

a. Evidentemente, para que o particionamento exista, B > 1.

b. Pelo teorema (III.1), B = $\Omega(\log n)$.

c. Pelo teorema (III.2), o tempo para calcular as tabelas ancestrais para todas as organizações possíveis com menos de B vértices é O (B . $2^{(3B - 4)}$). Pelo teorema (III.4), este tempo deve ser O (n), ou seja:

$$B \cdot 2^{(3B - 4)} = O(n)$$

Reunindo estas condições, GABOW & TARJAN [3] estimam o valor

$$B = \max \{ 2, \lfloor (\log (n / \log n)) / 3 \rfloor \}$$

Estão ilustrados na Tabela da figura (III.6) alguns valores de n com os valores de B correspondentes.

n	B
1024	2
2048	2
4096	2
8192	3
16384	3
32768	3
65536	4
131072	4

Alguns Valores de B
Figura III.6

III.8. Caso Particular da Árvore de Referência Degenerada.

Não é difícil antever que, devido ao laborioso procedimento inicial requerido antes de executar a sequência propriamente dita de T-UNIONs e T-FINDs, o algoritmo de Gabow, a despeito de sua complexidade linear, executará via de regra mais lentamente que os algoritmos alpha apresentados no Capítulo II, que dispensam tais cuidados iniciais.

Entretanto, um caso particular merece aqui ser ressaltado. A situação em que a árvore T se degenera em uma linha reta (e isto ocorre para uma série de aplicações) prescinde do procedimento de particionamento, uma vez que apenas uma organização estará presente (todos os microconjuntos terão B - 1 vértices, exceto porventura um deles que, ainda assim, pode ser completado com vértices fictícios). Neste caso, apenas o cálculo da tabela ancestrais para a organização em que cada um dos B - 1 vértices tem exatamente um filho (exceto o último) se faz necessário.

Observam ainda GABOW & TARJAN [3] que microfind (x), nestas circunstâncias, servirá apenas para localizar, examinando a máscara do microconjunto, o primeiro bite igual a zero em direção aos mais significativos a partir de uma posição dentro da máscara (correspondente ao vértice x). Em alguns processadores que possuem instruções para manipular cadeias de bites, isto pode ser conseguido com o uso de uma ou duas instruções de máquina, eliminando totalmente o cálculo da tabela ancestrais.

III.9. As Operações MACRO-UNION e MACRO-FIND.

Mostraremos agora a solução apresentada por AHO, HOPCROFT & ULLMAN [1] para a implementação das operações sobre macroconjuntos. O efeito de união por tamanho sem compressão de caminhos pode ser facilmente conseguido como é mostrado a seguir:

a. MACRO-FIND (c) é armazenada em um vetor *macfind* e leva, portanto, tempo O (1).

b. MACRO-UNION (a, b) determina, de início, as raizes r_a e r_b dos macroconjuntos que contêm (a) e (b) respectivamente (via macfind). Se o macroconjunto r_a é maior ou igual ao macroconjunto r_b, alteramos no vetor macfind apenas as posições correspondentes aos elementos de r_b, tornando-os elementos de r_a. Procedemos analogamente se r_b tem mais elementos do que r_a.

O algoritmo (III.6) implementa MACRO-UNION e MACRO-FIND. Repare que, para atualizar apenas as posições referentes aos elementos do menor conjunto em macfind, mantemos uma lista que se inicia na raiz de cada conjunto, encadeando seus elementos através do vetor *próx*. Inicialmente, para todo $v \in T$ que for raiz de microconjunto, fazemos: nome [v] = v, macfind [v] = v, tam [v] = 1 e próx [v] = 0.

Algoritmo III.6: Implementação de MACRO-FIND e MACRO-UNION.

```
func MACRO-FIND (c : inteiro) : inteiro;
início
    Retorne (nome [macfind [c]])
fim;

proc MACRO-UNION (a, b : inteiros);
var
    maior, menor, salva, r_a, r_b : inteiros;
início
    r_a := macfind [a];
    r_b := macfind [b];
    Se tam [r_a] ≥ tam [r_b] então
        maior := r_a;
        menor := r_b
    Caso contrário
        maior := r_b;
        menor := r_a;
        nome [r_b] := nome [r_a]
    fim;
    tam [maior] := tam [maior] + tam [menor];
    Enquanto menor ≠ 0 faça
        salva := próx [menor];
        macfind [menor] := maior;
        próx [menor] := próx [maior];
        próx [maior] := menor;
        menor := salva
    fim
fim;
```

Pelo Corolário II.1, o posto máximo de um vértice em uma floresta de w vértices, construída usando-se união por tamanho é log w. Como existem no máximo w - 1 MACRO-UNIONs possíveis, o tempo total para executá-los é O (w log w). Visto que cada um dos z MACRO-FINDs leva tempo O (1), a sequência total de até w - 1 MACRO-UNIONs e z MACRO-FINDs leva tempo O (z + w log w) para ser executada.

III.10. Considerações acerca da Implementação.

Todos os algoritmos citados neste capítulo foram implementados, constituindo um pequeno módulo que pode ser ligado ("link-editado") a programas de aplicação que utilizem as operações T-UNION e T-FIND. A propósito desta implementação, destacamos os seguintes aspectos que consideramos relevantes:

a. Foi utilizada a linguagem de programação C em um computador VAX 8810 com a versão 5.1 do sistema operacional VMS.

b. Estão disponíveis no módulo as seguintes rotinas:

inittree (nvértices): Prepara as estruturas necessárias ao particionamento da árvore T; deve ser a primeira rotina a ser chamada.

maketree (v, filhos): Sucessivas chamadas a esta rotina informam ao módulo a estrutura da árvore T; deve ser chamada uma vez para cada vértice que tenha pelo menos um filho.

partition (): Calcula B e particiona T em microconjuntos; deve ser chamada antes de começar a executar a sequência de operações.

link (v): Implementa T-UNION (v).

find (v): Calcula T-FIND (v).

c. Todas as estruturas são alocadas dinamicamente, evitando desperdícios com superdimensionamentos. Ao fim do particionamento, as estruturas auxiliares não utilizadas

pelas rotinas link e find são liberadas. Em particular, a matriz vértice foi implementada como uma coleção de vetores, já que o número de vértices varia de um microconjunto a outro.

d. A única limitação imposta por esta implementação concerne ao valor de B. Conforme mencionamos, as máscaras dos microconjuntos são armazenadas em números inteiros, designando-se um bite para cada vértice. Na linguagem C, dispomos de inteiros de 32 bites, o que sugere a limitação B < 33. Além disso, o valor máximo para a característica de um microconjunto é $(2^{eee-e} - 1) / 3$, que deve também caber em um inteiro de 32 bites, ou seja:

$$(2^{eee-e} - 1) / 3 < 2^{ee}$$

Isto impõe a restrição B < 18, o que é perfeitamente aceitável para qualquer valor prático de n, tendo em vista a estimativa da seção (III.7).

e. O algoritmo (III.5) (particionamento de T) foi implementado de maneira a evitar as chamadas ao procedimento calcula_ancestrais. Esta simplificação decorre imediatamente do fato de o valor B = 6 ser suficiente para qualquer aplicação prática. Em um programa a parte, calculamos todas as tabelas ancestrais para organizações com menos de 6 vértices. Este programa gerou como saída um outro programa em C, que contém a inicialização das tabelas em questão. Este último programa foi compilado e integrado ao módulo, evitando que as tabelas sejam calculadas durante o particionamento. Desta maneira, a condição imposta pelo teorema (III.4) perde o sentido e podemos adotar B = 6 para qualquer aplicação.

f. A título de ilustração, este programa que calcula as tabelas permitiu-nos obter o valor exato do somatório

$$S = \sum_{j=1}^{B-1} t_j . j . 2^j$$

proposto no teorema (III.2) que representa, além do tempo de cálculo, também o espaço ocupado pelas tabelas (em octetos). As tabelas da figura (III.7) resumem os resultados obtidos.

B	S
2	2
3	18
4	138
5	1034
6	7754
7	58442
8	442826
9	3371466

j	t_j
1	1
2	2
3	5
4	14
5	42
6	132
7	429
8	1430

Resultados Adicionais

Figura III.7

g. O caso particular mencionado na seção (III.8) dispensa qualquer particionamento ou cálculo de tabela e, nestas circunstâncias, o algoritmo pode competir em tempo de execução com os *algoritmos alpha* do Capítulo II.

CAPÍTULO IV

APLICAÇÃO: RECONHECIMENTO DE DIGRAFOS REDUTÍVEIS

Vamos examinar neste capítulo um problema cuja solução decorre da aplicação das operações UNION e FIND: o reconhecimento de digrafos de fluxo redutíveis através do algoritmo proposto por TARJAN em [10]. Inicialmente alguns conceitos da Teoria de Grafos serão revistos, com o objetivo de fixar a terminologia utilizada no capítulo. Seguem-se a idéia de Busca em Profundidade e alguns resultados a ela relacionados. Caracterizamos então os digrafos de fluxo redutíveis, apresentando o algoritmo de reconhecimento. Finalizamos com um quadro comparativo entre duas implementações do teste de redutibilidade: a que utiliza os algoritmos alpha do Capítulo II e a que utiliza o algoritmo de Gabow, do Capítulo III.

IV.1. Conceitos Iniciais e Busca em Profundidade.

Utilizaremos a terminologia adotada em [10].

Um *digrafo de fluxo* F é uma tripla (V, E, r), sendo (V, E) um digrafo e $r \in V$ tal que todo $w \in V$ é alcançável a partir de r. O vértice r é denominado *raiz* do digrafo de fluxo. Um vértice v *domina* um vértice w em um digrafo de fluxo F = (V, E, r) se v pertence a todo caminho de r a w.

Um digrafo de fluxo T = (V, E, r) é uma *árvore direcionada* se nenhuma aresta chega a r e qualquer outro vértice $V - \{ r \}$ tem apenas uma aresta chegando a ele. Se T é um subgrafo de um digrafo de fluxo F e T contém todos os vértices de F então T é uma *árvore de espalhamento* ou *árvore geradora* de F.

Entendemos como *busca* em um digrafo D = (V, E) qualquer processo sistemático para explorar D, caminhando por seus vértices e arestas. A princípio, nada é fixado quanto ao número de visitas que devem ser realizadas a cada vértice ou aresta, apenas que este número deve ser finito para que o algoritmo termine. Evidentemente, são preferíveis processos que visitem exatamente uma vez cada vértice e cada aresta de D.

SZWARCFITER [8] descreve um algoritmo genérico de busca. Entretanto, a busca em um digrafo não é única, uma vez que, no passo geral, devemos escolher um vértice marcado, isto é, já considerado anteriormente, e uma aresta inexplorada que parta deste vértice. O processo denominado *Busca em Profundidade* por TARJAN [9] fixa da seguinte forma a escolha do vértice marcado:

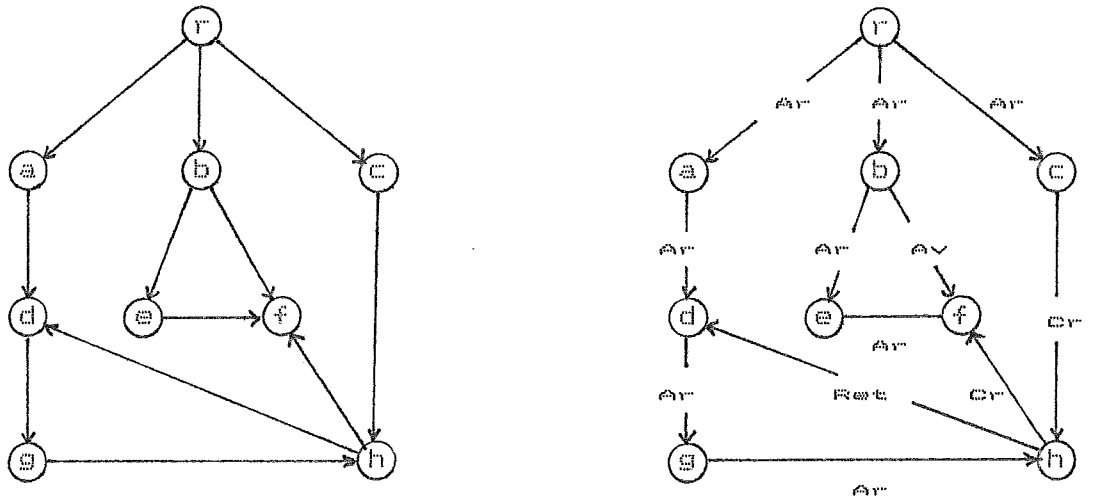
> *"Dentre os vértices marcados dos quais parte uma aresta inexplorada, escolher o mais recentemente alcançado na busca".*

Vista desta forma, a busca em profundidade constrói caminhos $Q = v_1, v_2, \ldots, v_{j-1}, v_j$ a partir de $v_1 = r$. No passo geral, se existe um vértice w tal que $(v_j, w) \in E$ e w nunca figurou em nenhum caminho (nunca foi alcançado), w é incluído em Q, que se torna $Q = v_1, v_2, \ldots, v_{j-1}, v_j, w$. Se não existir w satisfazendo a estas condições, v_j é retirado de Q e o mesmo processo é repetido para $Q = v_1, v_2, \ldots, v_{j-1}$. Logo, Q nada mais é do que uma pilha de vértices, contendo inicialmente o vértice de partida da busca (r).

Pode-se notar [9] que uma busca em profundidade realizada em um digrafo de fluxo F = (V, E, r) divide E em quatro subconjuntos disjuntos, classificando uma aresta (v, w) ∈ E em uma dentre quatro categorias. Imaginando que v está no topo de Q e que a aresta (v, w) vai ser explorada, ela pode ser classificada como aresta:

(i)	de *Árvore*	Quando w está desmarcado (w ainda não foi alcançado);
(ii)	de *Retorno*	Quando w está marcado e w ∈ Q (ou, analogamente, w está marcado e w alcança v por arestas de árvore);
(iii)	de *Avanço*	Quando w está marcado e v alcança w por arestas de árvore;
(iv)	de *Cruzamento*	Quando w está marcado e nem v alcança w nem w alcança v por arestas de árvore.

As arestas de árvore juntamente com suas extremidades constituem uma *Árvore de Espalhamento* para o digrafo de fluxo considerado, denominada *Árvore de Profundidade* [9]. Arestas de retorno são assim chamadas por atingirem vértices ainda empilhados, constituindo ciclos; a propósito, um digrafo de fluxo é acíclico se e somente se uma busca em profundidade não apresentar arestas de retorno. Arestas de avanço abreviam caminhos constituídos por arestas de árvore.



Exemplo de Busca em Profundidade

Figura IV.1

Para computar estes quatro subconjuntos na busca, Tarjan introduziu os conceitos de *Profundidade de Entrada (PE)* e *Profundidade de Saída (PS)* de um vértice, que significam,

respectivamente, a ordem em que determinado vértice é incluído e retirado de Q. A profundidade de entrada serve como a própria marca para o vértice, permanecendo igual a zero enquanto o vértice não tiver sido alcançado. A profundidade de saída é consultada para saber se o vértice ainda se encontra em Q (quando ela vale zero). Além disso, finda a busca e concluídos os cálculos de todas as profundidades, v alcança w por arestas de árvore se e somente se PE (v) $<$ PE (w) e PS (v) $>$ PS (w).

No algoritmo (IV.1) encontra-se implementada a busca em profundidade para um digrafo de fluxo F = (V, E, r). Observe que a pilha Q é construída implicitamente pela recursividade. O digrafo é armazenado na estrutura de adjacências *succ* [8].

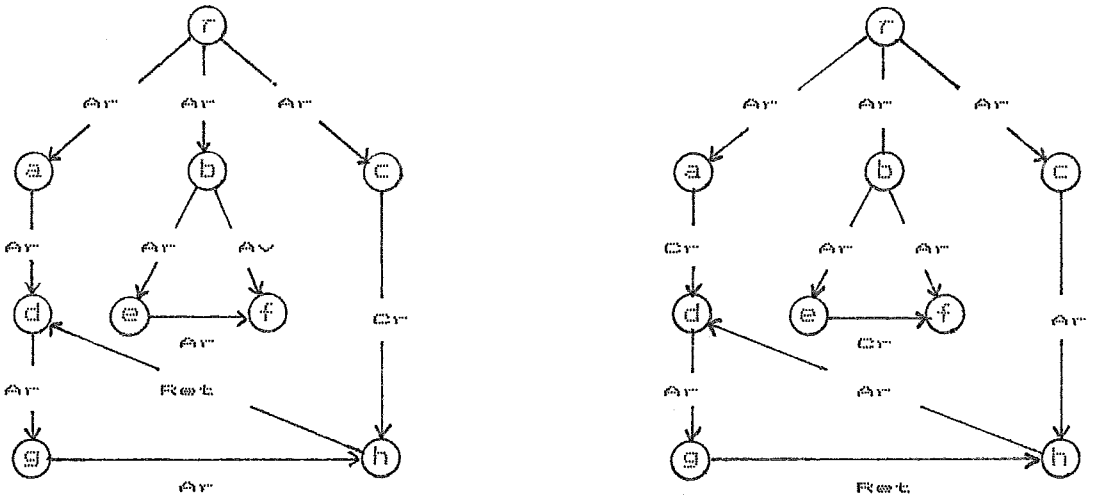
Algoritmo IV.1: Busca em Profundidade.

```
var
    pent, psai, r, n    : inteiros;
    PE, PS              : arranjos [1 .. n] de inteiros;
    succ, árvore, retorno,
    avanço, cruzamento  : arranjos [1 .. n] de listas;

proc busca_em_profundidade;
var
    v : inteiro;
início
    Para v := 1 até n faça
        PE [v]          := PS [v] := 0;
        árvore [v]      := lista_vazia;
        retorno [v]     := lista_vazia;
        avanço [v]      := lista_vazia;
        cruzamento [v] := lista_vazia
    fim;
    pent := psai := 0;
    bprof (r)
fim;
```

```
proc bprof (v);
var
    w : inteiro;
início
    PE [v] := pent := pent + 1;
    Para w ∈ succ [v] faça
        Se PE [w] = 0 então
            Inclua w em árvore [v];
            bprof (w)
        Caso contrário
            Se PS [w] = 0 então
                Inclua w em retorno [v]
            Caso contrário
                Se PE [v] < PE [w] então
                    Inclua w em avanço [v]
                Caso contrário
                    Inclua w em cruzamento [v]
                fim
            fim
        fim
    fim;
    PS [v] := psai := psai + 1
fim;
```

O algoritmo (IV.1) tem complexidade O (n + m) em tempo, pois cada aresta é visitada exatamente uma vez na busca [9].

Um fato conspícuo é que, mesmo fixado o vértice de partida, a classificação de arestas ainda assim é sensível à ordem dos vértices na estrutura de adjacências que representa o digrafo. Observe, na figura (IV.2), duas buscas realizadas sobre o mesmo digrafo: na primeira delas, a ordem dos vértices na lista de sucessores de r é a, b, c; na segunda, a ordem é c, b, a.



Sensibilidade da Busca à Ordenação dos Vértices Sucessores

Figura IV.2

IV.2. Caracterização dos Digrafos de Fluxo Redutíveis.

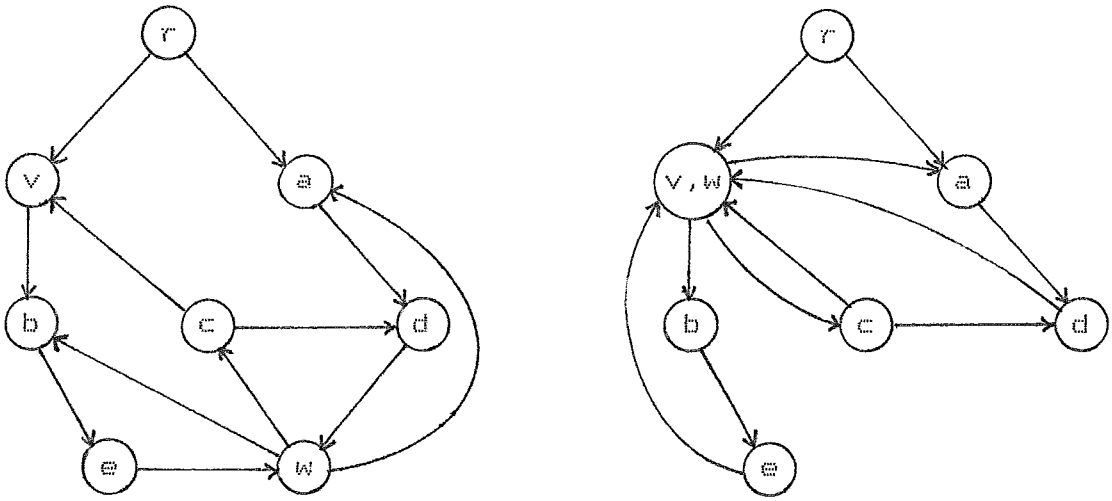
Seja $F = (V, E, r)$ um digrafo de fluxo e suponha que E não contém laços. Sejam $v, w \in E$, $w \neq r$. Denominamos *condensação de w em v* a operação que consiste em formar um novo digrafo de fluxo $F' = (V', E', r)$, onde:

$$V' = V - \{ w \}$$

$$E' = (E - \{ (x, w) \in E \} - \{ (w, x) \in E \}) \cup \{ (v, x) \mid x \neq v \text{ e } (w, x) \in E \} \cup \{ (x, v) \mid x \neq v \text{ e } (x, w) \in E \}$$

Ou seja, o novo digrafo tem um vértice a menos (w é excluído); quanto às arestas, retiramos todas as incidentes a w, acrescentando uma correspondente para cada aresta removida na qual extremidade w é trocada por v, evitando-se os laços.

Na figura (IV.3), mostramos um digrafo de fluxo e o digrafo dele resultante pela condensação do vértice w no vértice v.



Condensação de Vértices
Figura IV.3

Seja a transformação

Σ = Se (v, w) é a única aresta que chega a w ($w \neq r$), condense w em v.

HECHT & ULLMAN [5] demonstraram que a ordem na qual as transformações Σ são aplicadas não interferem no resultado; portanto, aplicando-se Σ sobre F enquanto possível obteremos sempre o mesmo resultado, que é um digrafo denominado *redução* de F.

Definição IV.1. Um digrafo de fluxo F = (V, E, r) é dito *redutível* se sua redução for o digrafo trivial F' = ({r}, {}, r).

Uma abordagem trivial para o teste de redutibilidade seria, a cada passo, localizar um vértice com grau de entrada 1 e condensá-lo com seu predecessor, gerando um novo digrafo no qual os graus de entrada são recalculados. Este passo seria repetido até que o digrafo fosse reduzido ao trivial ou não houvesse vértices com grau de entrada 1. Como a aplicação de Σ requer tempo O (n) e reduz de uma unidade o número de vértices, a complexidade em tempo do processo é $O(n^2)$.

HECHT & ULLMAN [5] propuseram várias caracterizações equivalentes para digrafos de fluxo redutíveis, uma das quais foi utilizada em [10] na concepção de um algoritmo mais eficiente para o reconhecimento. Sejam F = (V, E, r) um digrafo de fluxo sobre o qual foi realizada uma busca em profundidade, obtendo-se uma árvore de profundidade T = (V, E_T, r).

Lema IV.1. F é redutível se e somente se, para cada aresta de retorno (v, w) relativa a T, w domina v.

Esta caracterização possui forte analogia com a metodologia estruturada de programação. Se considerarmos a representação do fluxo do programa através de um digrafo, um laço no programa (proveniente de um comando do tipo *while*, *repeat*, *for*) corresponderia a uma aresta de retorno no digrafo. O lema (IV.1) diz que qualquer caminho de r a w deve passar por v, o que significa que v é o único ponto de entrada do laço em questão. Logo, programas estruturados têm fluxos representados por digrafos de fluxo redutíveis.

Um fato implícito no enunciado do lema (IV.1) e fundamental para sua correção é que o conjunto de arestas de retorno de um digrafo de fluxo redutível é invariante, ou seja, independe da ordenação dos vértices nas listas de sucessores, sendo o mesmo para qualquer busca em profundidade que se realize a partir de r. Na verdade, esta é uma caracterização adicional equivalente às anteriores [5].

IV.3. O Algoritmo de Reconhecimento.

Utilizando a caracterização do lema (IV.1), Tarjan constrói, para cada vértice w ∈ V, dois conjuntos:

Ret (w) = { v ∈ V | (v, w) é aresta de retorno }

Núcleo (w) = { v ∈ V ; existe z ∈ Ret (w) tal que
há um caminho de v a z que evita w }

Observe que Ret (w) ⊆ Núcleo (w).

O teorema (IV.1) a seguir, proposto por TARJAN [10], caracteriza os digrafos de fluxo redutíveis em função dos conjuntos Ret e Núcleo, calculados para todos os vértices.

Teorema IV.1. F é redutível se e somente se, para todo w ∈ V e para todo v ∈ Núcleo (w), w alcança v por arestas de árvore.

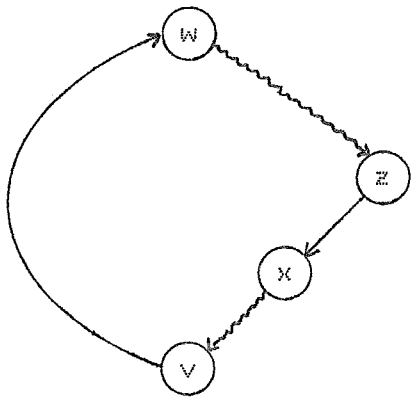
Com efeito, se existe w ∈ V e v ∈ Núcleo (w) tais que v não é descendente de w, então existe um caminho de r a algum vértice em Núcleo (w) que evita w; logo F não é redutível pelo lema (IV.1). Reciprocamente, se F não é redutível existe uma aresta de retorno (v, w) e um caminho de r a v que evita w. Então r ∈ Núcleo (w), mas r não é descendente de w.

Este resultado leva ao algoritmo de reconhecimento. Inicialmente, realizamos sobre F uma busca em profundidade, calculando, para cada vértice w ∈ V, as profundidades PE (w) e PS (w), bem como o conjunto Ret (w). No passo geral, seja w ∈ V o vértice sendo processado. Calculamos Núcleo (w). Havendo algum v ∈ Núcleo (w) que não seja descendente de w, o digrafo não é redutível. Caso contrário, os vértices em Núcleo (w) são condensados com w, formando um novo digrafo. Quanto à ordem em que os vértices devem ser considerados, TARJAN [10] demonstra que o processamento deve ser feito em ordem decrescente das profundidades de entrada dos vértices: o primeiro a ser processado é o último alcançado na busca, decrescendo até a raiz r do digrafo.

O cálculo de Núcleo (w) é simples. Inicialmente, Aux = Nucleo (w) = Ret (w). No passo geral, escolhemos x ∈ Aux e retiramos x de Aux. Analisamos todos os vértices z, predecessores de x, tais que (z, x) não seja aresta de

retorno. Se z ainda não tiver sido incluído em Núcleo (w) e z ≠ w, incluímos z em Aux e em Núcleo (w). A cada z considerado, aproveitamos para testar se w alcança z por arestas de árvore. Se isto não acontecer, o teste pode ser interrompido, pois o digrafo não será redutível. Repetimos este passo geral até que Aux se esvazie.

A figura (IV.4) mostra um passo da execução do algoritmo de redutibilidade.



Um Passo do Teste de Redutibilidade
Figura IV.4

É interessante notar que, se um digrafo F' resulta de F após uma sequência de condensações de vértices, vértices de F' correspondem a conjuntos de vértices de F. As operações UNION e FIND prestam-se então adequadamente para manipular estes conjuntos de vértices. O algoritmo (IV.2) mostra a implementação do Teste de Redutibilidade, conforme [10]. Observe que o procedimento de busca em profundidade separa as arestas que chegam a cada vértice em arestas de retorno (lista Ret) e as demais (lista Pred).

Algoritmo IV.2: Teste de Redutibilidade.

```
var
    PE, PS, Vértice : arranjos [1 .. n] de inteiros;
    pent, psai      : inteiros;
    n, r            : inteiros;
    Succ, Pred, Ret : arranjos [1 .. n] de listas de
                      inteiros;

proc Busca_em_Profundidade (v);
var
    w : inteiro;
início
    PE [v] := pent := pent + 1;
    Vértice [pent] := v;
    Para w ∈ succ [v] faça
        Se PE [w] = 0 então
            Inclua v em Pred [w];
            Busca_em_Profundidade (w)
        Caso contrário
            Se PS [w] = 0 então
                Inclua v em Ret [w]
            Caso contrário
                Inclua v em Pred [w]
            fim
        fim
    fim;
    PS [v] := psai := psai + 1
fim;


func redutível : lógica;
var
    Aux, Núcleo      : conjuntos de inteiros;
    u, w, x, y, z : inteiros;
início
    Para w := 1 até n faça
        Crie um conjunto unitário { w } cujo nome é w;
        PE [w]  := PS [w]   := Vértice [w] := 0;
```

```
        Ret [w] := Pred [w] := lista_vazia
    fim;
    pent := psai := 0;
    Busca_em_Profundidade (r);
    Para u := n descendo a 1 faça
        w := Vértice [u];
        Núcleo := {};
        Para x ∈ Ret [w] faça
            Núcleo := Núcleo U { FIND (x) };
        Aux := Núcleo;
        Enquanto Aux ≠ {} faça
            Escolha x ∈ Aux;
            Aux := Aux - { x };
            Para y ∈ Pred [x] faça
                z := FIND (y);
                Se PE [w] > PE [z] ou
                   PS [w] < PS [z] então
                    Retorne (FALSO);
                Se z ∉ Núcleo e z ≠ w então
                    Núcleo := Núcleo U { z };
                    Aux := Aux U { z }
                fim
            fim;
        fim;
        Para x ∈ Núcleo faça
            UNION (x, w, w);
    fim;
    Retorne (VERDADEIRO)
fim;
```

A complexidade de tempo é fácil de ser estimada: a busca em profundidade requer tempo O (n + m). Cada aresta de retorno é examinada exatamente uma vez, construindo o conjunto Núcleo inicial. Quando um vértice se torna um elemento de Núcleo, ele será condensado com algum outro vértice e seus predecessores jamais serão reexaminados; portanto, cada aresta que não seja de retorno também é examinada exatamente uma vez. Isto significa que a malha principal do algoritmo

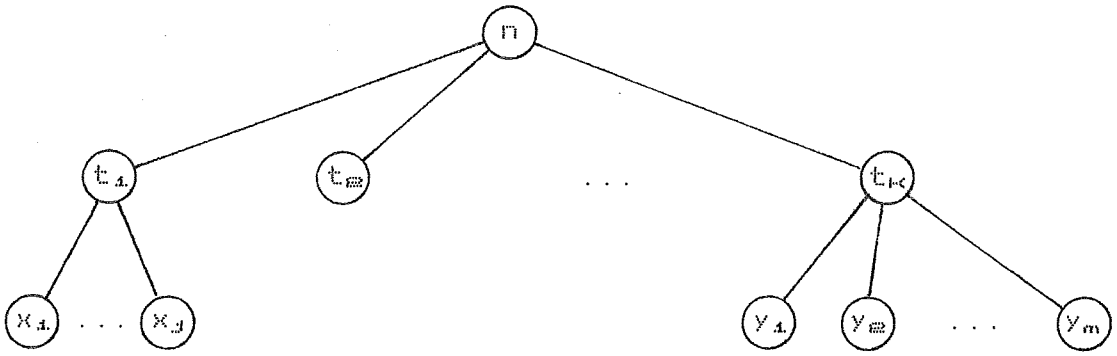
leva tempo O (n + m) mais o tempo para executar O (n) UNIONs e O (m) FINDs. Como m $\geq$ n, temos o tempo total O (m.α (m, n)), se utilizarmos os algoritmos *alpha*, ou O (m + n), se utilizarmos o algoritmo de Gabow para implementar UNION e FIND.

IV.4. Geração Pseudo-Aleatória de Digrafos de Fluxo Redutíveis.

Para estabelecer a desejada comparação entre os algoritmos alpha e o algoritmo de Gabow, vamos utilizar o teste de redutibilidade como aplicação para as operações UNION e FIND. Neste sentido, exploramos uma situação na qual a árvore de referência, que é a própria árvore de profundidade obtida na busca, não é necessariamente degenerada, obtendo um perfil do algoritmo de Gabow para casos mais gerais do que aqueles comentados por GABOW & TARJAN [3]. Para tanto, faz-se necessária a geração da massa de testes para os algoritmos, que deve ser constituída apenas por digrafos redutíveis, para que o teste não seja interrompido sem que todos os UNIONs e FINDs tenham sido executados.

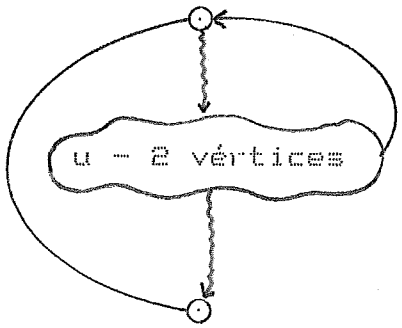
Como parâmetro para a geração fixamos o número de vértices do digrafo (n). Dividimos os n vértices em subconjuntos, da seguinte maneira: determinamos, de início, o tamanho K da partição, que é o número de subconjuntos que a compõem (fizemos K = $\lfloor \log n \rfloor$). A partição equânime teria subconjuntos de tamanho $\lfloor n / K \rfloor$ aproximadamente; geramos então um número inteiro aleatório t_1 no entorno de $\lfloor n / K \rfloor$ (mais precisamente no intervalo [n/2K, 3n/2K]) que será o tamanho do primeiro subconjunto. O problema agora se resume em dividir n - t_1 vértices em K - 1 subconjuntos, que é resolvido de maneira análoga. Ao fim, teremos K subconjuntos de tamanhos t_1, t_2, ..., t_K. Para cada um deles que possua tamanho superior a M (uma constante que fixamos em 20 vértices) repetimos o particionamento. O aspecto final é o

de uma árvore onde as folhas representam subconjuntos com até M vértices (figura (IV.5)).



Exemplo de Árvore de Particionamento
Figura IV.5

A idéia básica do algoritmo apóia-se na obtenção de *Núcleos Fortemente Conexos*, cujo formato é mostrado na figura (IV.6). Nestes núcleos existe uma aresta de retorno fundamental, ligando o último vértice ao primeiro; além disso, existe um caminho entre o primeiro vértice e cada vértice intermediário e um caminho entre cada vértice intermediário e o último. Portanto, os núcleos satisfazem aos requisitos do teorema (IV.1).



Formato de um Núcleo com u Vértices
Figura IV.6

Para gerar um núcleo com u $\leq$ M vértices, numeramo-los de 1 a u e geramos um digrafo acíclico com apenas uma fonte (o

vértice 1) e um único sumidouro (vértice u), acrescentando arestas (v, w), tais que 1 $\leq$ v < w $\leq$ u e também garantindo que todo vértice (exceto o primeiro) tenha grau de entrada 1 pelo menos. Em seguida, introduzimos alguns ciclos (eventualmente, nenhum) neste digrafo, mediante a adição de arestas da forma (v, 1), sendo 1 < v < u. No passo final, colocamos a aresta de retorno fundamental (u, 1).

O algoritmo de geração consiste então em um percurso em pós-ordem da árvore de particionamento: gerados os núcleos para todos os descendentes de um determinado nó da árvore, consideramos cada um deles como um único vértice condensado e tornamos a gerar um núcleo, até obtermos o digrafo completo, que corresponde à raiz da árvore.

O problema de considerar um núcleo como um só vértice sugere a seguinte questão: arestas destinadas ao (oriundas do) vértice condensado destinam-se a (originam-se de) que vértice do núcleo original? Pelo lema (IV.1), cada núcleo correspondente a uma aresta de retorno só deve ter um ponto de entrada, podendo ter vários de saída; portanto, arestas destinadas ao vértice condensado destinam-se, na verdade, ao primeiro vértice do núcleo original e arestas oriundas do vértice condensado podem ser oriundas de qualquer vértice do núcleo.

Constatamos, durante os testes realizados, que esta geração possui uma importante propriedade: para os digrafos gerados, o teste de redutibilidade efetua exatamente n - 1 UNIONs e m FINDs; isto é o ideal, uma vez que não estamos interessados no comportamento do teste de redutibilidade em função do tamanho do digrafo, mas na comparação entre duas implementações de UNION e FIND.

IV.5. Comparação de Resultados.

O algoritmo (IV.3) mostra como foram realizados os testes que nos permitiram comparar os algoritmos alpha com o algoritmo linear de Gabow.

Algoritmo IV.3: Comparação de Resultados.

```
início
    Para n := A até B passo C faça
        Repita X vezes
            Gere um digrafo com n vértices e m arestas
                tal que 3n - 100 < m < 3n + 100;
            Execute o algoritmo alpha para o digrafo;
            Execute o algoritmo de Gabow para o digrafo
        fim
    fim
fim;
```

Sabemos que o teste de redutibilidade, se for usado um algoritmo alpha para implementar UNION e FIND, executa em tempo O (m.α(m, n)). Como a geração dos digrafos é feita em função de n, procuramos manter o número de arestas m no entorno do valor 3n, de modo que, variando n linearmente, m também varie desta forma e, por conseguinte, as variações da expressão m.α(m, n) não sejam bruscas. Gerando X digrafos de n vértices e executando os algoritmos para cada um deles, obtemos uma faixa de variação do tempo de execução de ambos para cada valor de n. Tomamos ainda o cuidado de dividir o tempo de execução total em tempo de inicialização e tempo do algoritmo propriamente dito e, evidentemente, o teste de redutibilidade foi programado de maneira idêntica para ambos os casos. O algoritmo alpha empregado utiliza união por tamanho com compressão de caminhos.

Realizamos esta experiência para n variando de 1000 a 5500 em intervalos de 500 e gerando 30 digrafos para cada valor de n. Os resultados estão resumidos nas tabelas das figuras (IV.7) e (IV.8). Para cada valor de n, indicamos a faixa de variação dos valores de m (número de arestas), os tempos mínimo, médio e máximo de execução dos algoritmos e a razão G / A entre os tempos médios do algoritmo de Gabow e do algoritmo alpha.

	SEM INICIALIZAÇÃO						
	A L P H A			G A B O W			G / A
	min	méd	máx	min	méd	máx	
n = 1000							
m: 2905 a 3087	75	76	77	92	93.5	95	1.23
n = 1500							
m: 4452 a 4594	112	113.5	115	139	141	143	1.24
n = 2000							
m: 5946 a 6002	149	150	151	184	185.5	187	1.25
n = 2500							
m: 7433 a 7596	187	188.5	190	232	234	236	1.24
n = 3000							
m: 8925 a 9081	226	227	228	278	279.5	281	1.23
n = 3500							
m: 10432 a 10569	263	264.5	266	325	327	329	1.23
n = 4000							
m: 11943 a 12074	303	304	305	373	374.5	376	1.23
n = 4500							
m: 13428 a 13591	342	343.5	345	419	421.5	424	1.23
n = 5000							
m: 14928 a 15039	380	382	384	468	470	472	1.23
n = 5500							
m: 16471 a 16584	419	420.5	422	516	517.5	519	1.23

Comparação de Resultados

Figura IV.7

COM INICIALIZAÇÃO

	A L P H A			G A B O W			G / A
	min	méd	máx	min	méd	máx	
n = 1000							
m: 2905 a 3087	117	122.5	128	2041	2214	2387	18.07
n = 1500							
m: 4452 a 4594	189	192.5	196	4629	4861	5093	25.25
n = 2000							
m: 5946 a 6002	259	260	261	7249	7518	7787	28.91
n = 2500							
m: 7433 a 7596	325	328	331	9846	10142	10438	30.92
n = 3000							
m: 8925 a 9081	391	393.5	396	12458	12763	13068	32.43
n = 3500							
m: 10432 a 10569	459	462.5	466	15072	15414	15756	33.32
n = 4000							
m: 11943 a 12074	528	530.5	533	17676	18058	18440	34.04
n = 4500							
m: 13428 a 13591	595	598	601	20305	20637	20969	34.51
n = 5000							
m: 14928 a 15039	665	667	669	22917	23329	23741	34.97
n = 5500							
m: 16471 a 16584	736	737	738	25526	25713	25900	34.88

Comparação de Resultados

Figura IV.8

Examinando estes resultados, podemos observar os seguintes fatos:

a. O algoritmo alpha, no intervalo considerado para os testes, tem comportamento linear, como já havíamos previsto na seção (II.7) (α vale, no máximo, 4);

b. Se não considerarmos os tempos envolvidos em inicialização (figura (IV.7)), o algoritmo linear é cerca de 1.23 vezes mais lento que o algoritmo alpha utilizado;

c. Incluindo o tempo de inicialização (figura (IV.8)), o algoritmo de Gabow é, aproximadamente, 35 vezes mais lento que o algoritmo alpha, conforme também já havíamos suspeitado na seção (III.8).

CAPÍTULO V

OUTRAS APLICAÇÕES E CONCLUSÕES

Examinaremos, neste capítulo final, mais três aplicações para os algoritmos que implementam as operações UNION e FIND. Finalizamos com algumas conclusões a respeito do trabalho realizado.

V.1. Outras Aplicações para as Operações UNION e FIND.

Além do algoritmo para reconhecimento de digrafos de fluxo redutíveis examinado no Capítulo IV, as operações UNION e FIND são ainda aplicáveis na solução dos seguintes problemas:

a. Equivalência de Autômatos Finitos. [1]

Definição V.1. Um *Autômato Finito Determinístico (AFD)*, ou simplesmente autômato finito, é uma quíntupla $M = (S, I, \Delta, s_0, F)$, onde

. S é um conjunto finito de *estados*;
. I é um conjunto finito, denominado *alfabeto de entrada*;
. Δ: S x I -> S, denominada *função de transição*;
. $s_0 \in S$ é o *estado inicial*;
. $F \subseteq S$ é o conjunto de *estados finais*.

Se denotarmos por I^* o conjunto (infinito) de todas as cadeias de comprimento finito construídas com elementos de I (inclusive a cadeia vazia ϵ), podemos estender Δ da seguinte forma:

Δ^*: S x I* -> S

Δ^* (s, ϵ) = s
Δ^* (s, xa) = Δ (Δ^* (s, x), a), x $\in$ I*, a $\in$ I.

Uma cadeia x $\in$ I* é aceita por M se e somente se Δ^* (s_0, x) $\in$ F. A linguagem aceita por M, denotada por L (M), é o conjunto de todas as cadeias de I* aceitas por M.

Definição V.2. Dizemos que dois estados s_1 e s_2 são equivalentes se, para todo x $\in$ I*, Δ^* (s_1, x) $\in$ F <=> Δ^* (s_2, x) $\in$ F. Dois autômatos finitos são equivalentes se aceitam a mesma linguagem.

Para decidir se $M_1 = (S_1, I, \Delta_1, s_1, F_1)$ e $M_2 = (S_2, I, \Delta_2, s_2, F_2)$ são equivalentes, utilizamos as operações UNION e FIND como mostra o algoritmo (V.1). Vamos supor, sem perda de generalidade, que S_1 e S_2 são disjuntos. As operações UNION e FIND são utilizadas para manipular uma partição do conjunto $S_1 \cup S_2$, de maneira que estados equivalentes situem-se em um mesmo bloco. Para isto, basta observarmos que, se dois estados s_1 e s_2 são equivalentes, para todo a $\in$ I, Δ (s_1, a) e Δ (s_2, a) também o são. Além disso, um estado final jamais pode ser equivalente a um estado não-final, graças à cadeia vazia ϵ.

Ao fim do algoritmo (V.1), os conjuntos formados representam uma partição de $S_1 \cup S_2$ em blocos de estados que devem ser equivalentes. M_1 e M_2 são equivalentes se e somente se a nenhum bloco pertencem, simultaneamente, um estado final e um estado não-final.

Algoritmo V.1: Separação dos Estados em Blocos de Equivalência.

início
    Inicialize Lista com o par (s_1, s_2);
    Para $s \in S_1 \cup S_2$ *faça*
        Crie o conjunto { s } chamado s;
    Enquanto Lista não *for* vazia *faça*
        Retire um par (x, y) de Lista;
        X := FIND (x); Y := FIND (y);
        Se $X \neq Y$ *então*
            UNION (X, Y, X);
            Para $a \in I$ *faça*
                Inclua (Δ_1 (x, a), Δ_2 (y, a)) em Lista
        fim
    fim
fim;

b. Problema do Mínimo Fora-de-Linha. [1]

Seja um conjunto $S \subseteq \{ x \in \mathbb{N} \mid 1 \leq x \leq n \}$, inicialmente vazio, e duas operações definidas sobre S:

a. INSERE (S, i): cujo efeito é acrescentar i a S, sendo $1 \leq i \leq n$;
b. MÍNIMO (S): cujo efeito é determinar o menor elemento de S, retirando-o do conjunto.

Consideremos uma sequência W de operações INSERE e MÍNIMO sendo que, para todo i tal que $1 \leq i \leq n$, INSERE (S, i) aparece no máximo uma vez. O problema conhecido como mínimo fora-de-linha consiste em determinar a sequência de inteiros retirados de S pela operação MÍNIMO sem executar a sequência W, que é conhecida *a priori*.

Suponhamos que existam, na sequência W, k instruções MÍNIMO. Logo, W tem a seguinte estrutura:

$$I_1 \; M \; I_2 \; M \; \ldots \; M \; I_k \; M \; I_{k+1}$$

onde as subsequências I_j, $1 \leq j \leq k+1$, contêm apenas instruções INSERE e M são as instruções MÍNIMO. Vamos simular W com o uso das operações UNION e FIND. Os conjuntos da coleção são inicializados da seguinte maneira: o conjunto de nome j ($1 \leq j \leq k+1$) contém o elemento i se a operação INSERE (i) pertencer à subsequência I_j. Dois arranjos *pred* e *succ* formam uma lista duplamente encadeada com os valores j para os quais existe um conjunto com o nome j. O algoritmo (V.2) mostra a solução. Observe que, além do arranjo *nome*, os demais arranjos *pai*, *tam* e *raiz*, necessários à execução de UNION e FIND (ver algoritmo (II.6)), são também inicializados de modo a refletir corretamente a partição inicial, em que os conjuntos não são unitários como de costume.

Algoritmo V.2: Solução para o Problema do Mínimo Fora-de-Linha.

```
início
    Para j := 1 até k + 1 faça
        p := primeiro elemento de I_j;
        nome [p] := j;
        raiz [j] := p;
        tam  [p] := 1;
        pai  [p] := 0;
        Para os demais elementos i de I_j faça
            pai [i] := p;
            raiz [i] := 0;
            tam [p] := tam [p] + 1
        fim;
        pred [j] := j - 1;
        succ [j - 1] := j
    fim;
```

```
    Para i := 1 até n faça
        j := FIND (i);
        Se j < k então
            Imprima (i, " é removido pela ", j,
                        "-ésima operação MÍNIMO");
            UNION (j, succ [j], succ [j]);
            succ [pred [j]] := succ [j];
            pred [succ [j]] := pred [j]
        fim
    fim
fim;
```

c. Árvore Geradora Máxima. [8]

Consideremos um grafo $G = (V, E)$ no qual as arestas $e = (v, w) \in E$ possuem um peso $p(e)$ associado. Denominamos peso de uma árvore geradora $T = (V, E_T)$ de G à soma dos pesos das arestas de E_T. Desejamos obter, dentre todas as árvores geradoras de G, uma que tenha peso máximo, ou seja, obter E_T tal que

a. (V, E_T) seja uma árvore;

b. $\sum_{e \in E_T} p(e)$ seja máximo.

A solução consiste na aplicação de um algoritmo guloso cuja correção é provada em [8]. Inicialmente, tomamos $E_T = \emptyset$ e, passo a passo, escolhemos uma aresta $(v, w) \in E - E_T$ tal que sua inclusão em E_T não introduza ciclos na árvore T sendo formada e o peso total de $E \cup \{ (v, w) \}$ seja o maior possível. As operações UNION e FIND prestam-se a evitar a formação de ciclos, como é mostrado no algoritmo (V.3).

Algoritmo V.3: Árvore Geradora Máxima.

```
início
    Ordenar E não-crescentemente por pesos;
    Para v ∈ V faça
        Crie um conjunto { v } de nome v;
    E_T := ∅;
    Para i := 1 até m faça
        Seja e_i = (v, w);
        A := FIND (v);
        B := FIND (w);
        Se A ≠ B então
            UNION (A, B, A);
            E_T := E_T ∪ { (v, w) }
        fim
    fim
fim;
```

V.2. Conclusões.

Esta tese reuniu uma série de resultados sobre algoritmos que manipulam conjuntos e, mais enfaticamente, coleções de conjuntos disjuntos.

Em um primeiro momento, detivemo-nos com o estudo do *tipo abstrato conjunto* e de duas implementações para ele, usadas com frequência por linguagens de programação que oferecem em seus repertórios tal tipo de dados. Abordamos então um problema que serviu como motivação inicial para o desenvolvimento de algoritmos que operam sobre coleções de conjuntos disjuntos: o problema dos pares equivalentes. De imediato mostramos a representação de coleções através de florestas direcionadas e destacamos as duas operações, UNION e FIND, usualmente definidas sobre uma coleção.

O Capítulo II ocupou-se em formalizar estas operações,

desenvolver algoritmos que as implementassem eficientemente e calcular a complexidade de tempo para executar uma sequência constituída de UNIONs e FINDs intercalados. Iniciamos com uma formalização algébrica do problema, definindo as operações LINK, EVAL e UPDATE das quais UNION e FIND tornam-se um caso particular. Obtivemos uma complexidade expressa em termos de α, uma função de crescimento extremamente lento relacionada ao inverso da função de Ackermann. Comentamos a inexistência de um algoritmo linear para o problema geral, tomando como base o modelo computacional denominado *pointer machine*, no qual o problema possui limite inferior expresso também em termos de α, o que atesta a otimalidade dos *algoritmos alpha* mencionados.

Para um caso particular do problema, no qual a árvore de uniões é conhecida antes mesmo de a sequência ser executada, analisamos no Capítulo III uma solução que aliou os algoritmos já conhecidos a buscas em pequenas tabelas; o algoritmo resutante tem complexidade linear. Apesar disto, apenas para árvores degeneradas de uniões este algoritmo linear torna-se competitivo com os tradicionais *algoritmos alpha*; em se tratando de uma árvore não-degenerada, um laborioso (embora linear) procedimento de divisão da árvore em microconjuntos se faz necessário e o algoritmo executa, via de regra, mais lentamente que os anteriores.

Esta afirmação encontra-se ilustrada em uma comparação que estabelecemos entre os algoritmos no Capítulo IV, usando o reconhecimento de digrafos de fluxo redutíveis como aplicação. Neste caso, a árvore de uniões coincide com a árvore de profundidade do digrafo, mas não é necessariamente degenerada. Para esta comparação, descrevemos o método utilizado na geração pseudo-aleatória dos digrafos e apresentamos os resultados decorrentes dos testes realizados. Finalmente, comentamos mais algumas aplicações para as operações UNION e FIND.

Queremos destacar a relevância do assunto abordado nesta

monografia. A complexidade para execução de uma sequência de UNIONs e FINDs foi tema de inúmeros artigos publicados desde 1973 até o fim da década de 80. Isto se deve ao fato de tais operações constarem das soluções de diversos problemas computacionais. Além das aplicações mencionadas nesta tese, GABOW & TARJAN [3] citam ainda as seguintes: escalonamento em um sistema de dois processadores, emparelhamento em grafos convexos, roteamento de canais em circuitos VLSI, problema dos ancestrais comuns em árvores, árvores de espalhamento com interseção mínima de arestas e separadores para grafos cordais.

Ressaltamos mais uma vez o caráter essencialmente didático deste trabalho: procuramos estabelecer uma sequência coerente para enunciar os resultados pesquisados na literatura, dispersos em artigos, heterogêneos em notação e, por vezes, obscuros; tornamos as demonstrações bastante claras, sobretudo o cálculo amortizado que resulta na complexidade α; por fim, incluímos em pseudo-código facilmente implementável em qualquer linguagem estruturada de programação no estilo de Pascal ou C todos os algoritmos que mencionamos.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] AHO, A.V., HOPCROFT, J.E. & ULLMAN, J.D. The Desing and Analysis of Computer Algorithms. Addison-Wesley, Reading, Mass., 1974.

[2] BANACHOWSKI, L. A Complement to Tarjan's Result about the Lower Bound on the Complexity of the Set Union Problem. Information Processing Letters, vol. 11, nº 2, pp. 59-65, Oct. 1980.

[3] GABOW, H.N. & TARJAN, R.E. A Linear Time Algorithm for a Special Case of Disjoint Set Union. Proc of the 15th ACM Symposium on Theory of Computing, pp. 246-251, 1983.

[4] GALLER, B.A. & FISHER, M.J. An Improved Equivalence Algorithm. Comm of the ACM, vol. 7, nº 5, pp. 301-303, May 1964.

[5] HECHT, M.S. & ULLMAN, J.D. Flow Graph Reducibility. SIAM Journal on Computing, vol. 1, pp. 188-202, 1972.

[6] HOPCROFT, J.E. & ULLMAN, J.D. Set Merging Algorithms. SIAM Journal on Computing, vol. 2, nº 4, pp. 294-303, Dec. 1973.

[7] HOROWITZ, E. & SAHNI, S. Fundamentos de Estruturas de Dados. Rio de Janeiro, Ed. Campus, 2ª Edição, 1986.

[8] SZWARCFITER, J.L. Grafos e Algoritmos Computacionais. Rio de Janeiro, Ed. Campus, 1984.

[9] TARJAN, R.E. Depth-First Search and Linear Graph Algorithms. SIAM Journal on Computing, vol. 1, Dec. 1972.

[10] TARJAN, R.E. Testing Flow Graph Reducibility. Journal of Computer and System Sciences, vol. 9, nº 3, pp. 355-365, Dec. 1974.

[11] TARJAN, R.E. Efficiency of a Good but not Linear Set Union Algorithm. Comm. of the ACM, vol. 22, nº 2, pp. 215-225, Apr. 1975.

[12] TARJAN, R.E. A Class of Algorithms which Require Non-Linear Time to Maintain Disjoint Sets. Journal of Computer and System Sciences, vol. 18, nº 2, pp. 110-127, Apr. 1979.

[13] TARJAN, R.E. Applications of Path Compression on Balanced Trees. Journal of the ACM, vol. 26, nº 4, pp. 690-715, Oct. 1979.

[14] TARJAN, R.E. Data Structures and Network Algorithms. CBMS Regional Conference Series in Applied Mathematics 44. SIAM, Philadelphia, Penn., 1983.

[15] TARJAN, R.E. & VAN LEEUWEN, J. Worst-Case Analysis of Set Union Algorithms. Journal of the ACM, vol. 31, nº 2, pp. 245-281, Apr. 1984.

[16] TARJAN, R.E. Amortized Computational Complexity. SIAM Journal on Algebraic and Discrete Mathematics, vol. 6, nº 2, pp. 306-318, Apr. 1985.

[17] VAN LEEUWEN, J. & VAN DER WEIDE, T. Alternative Path Compression Techniques. Tech. Rep. RUU-CS-77-3, Rijksuniversiteit Utrecht, Utrecht, the Netherlands, 1977.