

MÉTODOS AUTOMÁTICOS DE

GERÊNCIA DE MEMÓRIA

JORGE STOLFI

DISSERTAÇÃO APRESENTADA AO

INSTITUTO DE MATEMÁTICA E ESTATÍSTICA

DA

UNIVERSIDADE DE SÃO PAULO

PARA OBTENÇÃO DO GRAU DE MESTRE

EM

MATEMÁTICA APLICADA

ORIENTADOR

Prof. Dr. TOMASZ KOWALTOWSKI.

- São Paulo, julho de 1979 -

ao Augier e ao Geraldo,
por essa doença chamada Computação.

Meus Agradecimentos

ao Prof. Dr. Tomasz Kowaltowski, pela paciente e valiosa orientação;

aos Profs. Drs. Valdemar Waingort Setzer, Imre Simon, Ivan de Queiroz Barros e Cyro de Carvalho Patarra, pelo apoio recebido como docente e aluno deste Departamento;

aos colegas e alunos do Instituto, pelo constante estímulo;

e a minha esposa, pela compreensão, incentivo e inestimável auxílio,

sem os quais este trabalho não existiria.

ABSTRACT

The object of this work is to describe and analyze several techniques for the automatic management of memory space, in computer systems and languages that allow linked data structures and liberal allocation of storage areas.

We are mainly concerned with the recovery of inaccessible storage areas (better known as "garbage collection") since this is usually the most complex and demanding function of any memory manager that doesn't rely on user-driven recovery.

A special algorithmic notation is developed here for the description of memory management methods.

First, we study several algorithms that traverse and mark all accessible records of a data structure, and we classify them according to their behaviour. These algorithms are used in garbage collectors of the "mark and collect" type, that are described next. This class includes the "garbage compactors", that relocate accessible records in order to coalesce all available space in a single area.

The second class of garbage collectors studied consists of methods that copy all accessible records to a fresh memory space, and reclaim the old.

Finally, "on-the-fly" garbage collectors, that can be run in parallel with the user program, are described and analyzed.

Special attention is paid throughout for data structures with overlapping and/or very small nodes, since they are very common in modern high-level languages, and pose serious problems garbage collectors.

ÍNDICE

INTRODUÇÃO 1

CAPÍTULO I — NOÇÕES FUNDAMENTAIS

- 1.1 Variáveis 5
- 1.2 Memória, Áreas e Endereços 6
- 1.3 Memória Paginada 8
- 1.4 Registros e Campos 9
- 1.5 Gerência de Memória 12
- 1.6 Referências e Apontadores 13
- 1.7 Variáveis Ativas e Inativas 15
- 1.9 Cadeias, Estruturas e Árvores 18
- 1.10 Classificação dos Recuperadores

CAPÍTULO II — A REPRESENTAÇÃO DOS ALGORITMOS

- 2.1 Generalidades 24
- 2.2 Condicionais, Repetições e Blocos 24
- 2.3 Áreas, Modos e Declarações 25
- 2.4 Modos Inteiro, Booleano e Dado 27
- 2.5 Modo Endereço 28
- 2.6 União de Modos 28
- 2.7 União Uniforme 29
- 2.8 Modos Compostos e Sequências 31
- 2.9 Procedimentos 32
- 2.10 Valores Simbólicos e Modos Degenerados 33
- 2.11 Outras Características da Notação 34
- 2.12 Análise da Eficiência dos Algoritmos 35
- 2.13 Definições de Modos e Áreas de Uso Geral 37

CAPÍTULO III — ALGORITMOS DE PERCURSO E MARCAÇÃO

- 3.1 Introdução 39
- 3.2 Ordem de Marcação e Árvore de Percurso 40
- 3.3 Algoritmos Horizontais 41

3.4	Espaço Usado pelo Algoritmo	44
3.5	Otimização do Uso da Pilha	47
3.6	Otimização do Número de Falhas de Páginas	49
3.7	Marcação com Pilha Distribuída	52
3.8	Marcação com Pilha Insuficiente	54
3.9	Algoritmos Verticais	56
3.10	A Variante de Clark	59
3.11	O Algoritmo de Deutsch, Schorr e Waite	61
3.12	Marcação de Nós Amorfos e Sobrepostos	65
3.13	Marcação por Segmentos	69
3.14	O Algoritmo Marcador de Zave	72
3.15	Outros Algoritmos de Marcação	74
3.16	Algoritmos de Marcação: Conclusões	78

CAPÍTULO IV – MÉTODOS DE COLETA E COMPACTAÇÃO

4.1	Coleta de Nós de Tamanho Uniforme	81
4.2	Coleta de Nós de Tamanho Variado	83
4.3	Algoritmos de Compactação	87
4.4	Compactação por Tabela de Relocação	90
4.5	Compactação com Tabela nos Interstícios	92
4.6	Compactação de Nós Uniformes	96
4.7	Compactação de Estruturas com Poucos Nós Ativos	99
4.8	Compactador Genérico Sem Tabela	101
4.9	Compactação com Nós e Campos Amorfos	105
4.10	O Compactador de Zave	108

CAPÍTULO V – RECUPERADORES POR CÓPIA

5.1	Introdução	111
5.2	Cópia com Marcação Prévia	113
5.3	Cópia com Fila	115
5.4	O Algoritmo de Cheney	117
5.5	Algoritmos de Cópia Verticais	118
5.6	O Algoritmo de Reingold	119
5.7	O Copiador de Clark	121
5.8	Nós Amorfos e Nós Sobrepostos	125

CAPÍTULO VI — RECUPERADORES EM TEMPO REAL

- 6.1 Introdução 128
- 6.2 Sincronização de Processos 129
- 6.3 Sincronização Eficiente entre Recuperador e Usuário 132
- 6.4 O Algoritmo de Steele 134
- 6.5 Justificativa do Algoritmo de Steele 145
- 6.6 Análise do Recuperador de Steele 151
- 6.7 O Algoritmo de Baker 155
- 6.8 Análise do Algoritmo de Baker 163
- 6.9 Técnicas Para Aumentar o Paralelismo 169

CAPÍTULO VIII — CONCLUSÕES FINAIS

- 7.1 Comparação dos Métodos de Recuperação 169
- 7.2 Pesquisas em Andamento e Problemas em Aberto 171

BIBLIOGRAFIA 176

INTRODUÇÃO

Este trabalho é uma descrição dos principais algoritmos para a recuperação de espaço inativo em sistemas com alocação dinâmica de memória.

Tais algoritmos são conhecidos na literatura em inglês como "garbage collectors" (literalmente "coletores de lixo"), por localizarem as áreas da memória utilizadas e descartadas por um programa (isto é, o "lixo") e recolherem-nas a um estoque de áreas livres, de onde elas podem ser extraídas e re-usadas pelo mesmo programa, quando necessário.

Procuramos dar maior ênfase neste trabalho aos aspectos práticos do problema, e nos preocupamos, na medida do possível, com detalhes da implementação dos algoritmos e do meio onde eles usualmente serão executados. Isso nos obrigou a sacrificar em alguns pontos a precisão matemática e o aprofundamento dos problemas teóricos.

Em particular, muitas das definições adiante encontradas são incompletas e vagas, para os padrões matemáticos; os termos "quase", "geralmente", "aproximadamente", e similares, ocorrem talvez com frequência excessiva. Entretanto, considerando-se que os conceitos aqui tratados são oriundos do mundo real, e portanto naturalmente vagos e incompletos, acreditamos que tais definições e afirmações são suficientemente precisas para serem usáveis com proveito.

Os algoritmos não são examinados em ordem cronológica, mas agrupados segundo suas características de estrutura e funcionamento. Embora o material tenha sido quase que total-

mente obtido da literatura existente, este trabalho pretende ser uma análise de métodos, e não de artigos.

Assim, todos os algoritmos foram reescritos, com mo dificações mais ou menos profundas, de modo a revelar ao má ximo as semelhanças e diferenças em suas estruturas. Alguns deles foram desmembrados, sendo suas partes analisadas como algoritmos distintos, enquanto outros foram simplificados ou agregados.

Considerada a exigüidade de espaço, decidimos omitir quase que totalmente as demonstrações dos algoritmos, e a de rivação das fórmulas que exprimem o tempo e espaço auxiliar exigidos pelos mesmos. Apenas nos casos mais complexos ou de licados, indicamos, sucinta e informalmente, os princípios es senciais da demonstração.

Acreditamos ser essa omissão justificável, em vista da simplicidade dos algoritmos, e do fato que as referências originais dos mesmos contem frequentemente tais demonstrações. O valor prático das mesmas é, além disso, bastante questio na vel: sabe-se que a demonstração manual de um algoritmo está também sujeita a erros, talvez mais que o próprio algoritmo, e teria que ser totalmente refeita em cada implementação des te.

Preferimos desenvolver uma notação própria para os algoritmos, em vez de recorrer a linguagens de programação e xistentes. Fomos nisso motivados pelas restrições que as lin guagens de alto nível impõem à manipulação de endereços e ã reas de memória, e pela existência nelas de mecanismos de ge rência de memória excessivamente sofisticados, cuja implemen ta ção é justamente o objeto de nossos estudos.

O trabalho consiste de sete capítulos, além desta in trodução. No primeiro, definimos, ou precisamos, vários ter-

mos e expressões que serão utilizados no decorrer do trabalho; os mesmos são sublinhados na ocorrência que os define. Deve-se observar que a nomenclatura dos conceitos aqui estudados não é universal, e a nossa pode diferir em pontos essenciais da usada por outros autores.

O capítulo II é uma descrição da notação utilizada para descrever os algoritmos, e a definição dos parâmetros u utilizados na análise de sua eficiência.

No capítulo III, apresentamos os algoritmos fundamentais para o percurso e marcação das áreas potencialmente em uso. Como esses algoritmos são fundamentais para a tarefa de recuperação de espaço, dedicamos especial atenção a sua análise.

O assunto do capítulo IV são os recuperadores por coleta e compactação, que exigem a execução prévia de um dos algoritmos do capítulo III, e recolhem as áreas não marcadas pelo mesmo ao estoque de áreas disponíveis. Os recuperadores por compactação reorganizam nesse processo o conteúdo da memória, de modo a transformar o espaço disponível numa área contínua.

No capítulo V, tratamos de outra classe de recuperadores, que copia as áreas potencialmente usáveis para novas posições consecutivas na memória, e incorpora as áreas originais (usáveis ou não) ao espaço livre.

O capítulo VI descreve algoritmos e técnicas utilizadas para executar a recuperação simultaneamente com o programa usuário, evitando a suspensão deste por períodos muito longos.

Finalmente, apresentamos no capítulo VII algumas comparações entre as várias técnicas examinadas, e relacionamos

alguns dos problemas em aberto e áreas de pesquisa relacionados com o assunto deste trabalho.

CAPÍTULO I - NOÇÕES FUNDAMENTAIS

1.1 Variáveis

O conceito de variável existe em quase todas as linguagens de programação: um objeto que, durante o processamento de um programa, contém a cada instante uma informação, ou valor.

A operação de consulta a uma variável produz seu valor corrente; a atribuição substitui este por outro valor dado. Geralmente, variáveis podem ser criadas e destruídas no decorrer da execução de um programa.

Uma variável pode ser composta, isto é, formada por duas ou mais variáveis, seus componentes, cada um dos quais pode ser manipulado independentemente dos demais. Um componente de uma variável composta pode ser ele próprio uma variável composta. Componentes podem ser obtidos aplicando-se operações de seleção ou indexação às variáveis que os contem.

Uma variável que não pode ser subdividida em compontes é dita simples. Em quase todas as linguagens de alto nível, duas variáveis simples distintas são sempre independentes, isto é, a atribuição de um valor a uma delas não afeta o da outra.

Toda variável composta é no fundo formada por compontes simples. Duas variáveis compostas distintas podem não ser independentes; neste caso, elas possuem um ou mais componentes simples em comum.

Os valores armazenáveis numa variável estão sempre restritos, desde o momento de sua criação, a um conjunto finito (o tipo da variável), definido pelo programador ou pelas regras da linguagem.

Em algumas linguagens, as variáveis podem ser difíceis de reconhecer. Em LISP 1.5 {Car 4/60, Wei 67}, por exemplo, as variáveis podem ser identificadas com as "células" que constituem as S-expressões, e com os elementos das P-listas e da A-lista que contem os valores dos "átomos". A atribuição nas primeiras é feita pelas funções RPLACA e RPLACD, e nas demais por SET, CSET, DEFINE e pelo mecanismo de passagem de parâmetros.

Em ALGOL 68 {Wij 75}, os objetos com "modo" ref μ podem ser interpretados como variáveis, pois podem lhes ser atribuídos valores do "modo" μ .

Além disso, consideraremos variáveis todos os temporários, acumuladores, registradores, etc., criados e manipulados implicitamente pelo compilador para armazenar resultados intermediários de operações complexas, ou "constantes" (como as do ALGOL 68) cujo valor é calculado no decorrer da execução do programa.

1.2 Memória, Áreas e Endereços

A semântica de uma linguagem de programação de alto nível é usualmente descrita em termos de objetos abstratos, como variáveis, valores inteiros, reais, etc.. A implementação de uma linguagem num computador específico exige que se estabeleça uma correspondência entre esses objetos abstratos e os objetos existentes neste último.

Uma parte essencial de todo sistema de processamento de dados é um dispositivo de armazenamento de informações, ou memória. Em quase todos os sistemas, esta assume a forma de uma memória linear, constituída de uma sequência finita de unidades elementares; indivisíveis e uniformes, cada uma das quais, em cada instante, pode assumir um de um conjunto finito de estados possíveis. Uma sequência finita de tais estados será chamada de dado.

Na maioria dos casos, cada unidade elementar pode assumir apenas dois estados, sendo chamada de dígito binário ou bit. Entretanto, pode ser conveniente as vezes considerar a memória subdividida em unidades maiores ("dígitos decimais", "caracteres", "palavras", etc.) e tratar tais unidades como elementares.

Cada unidade elementar é identificada por um endereço, único e exclusivo. Neste trabalho, vamos supor que o conjunto dos endereços é um intervalo dos números inteiros.

A ordem natural dos endereços induz uma ordem sobre as unidades elementares; podemos portanto dizer que uma precede ou segue outra, ou que duas delas são consecutivas, e assim por diante, se essas afirmações valerem para seus endereços.

Uma coleção de unidades elementares será chamada de espaço de memória. O tamanho do espaço S , denotado por $\text{tam}(S)$, é o número de unidades elementares em S . O conteúdo de um espaço de memória num dado instante é o dado formado pela sequência dos estados correntes de suas unidades, na ordem natural destas. Dois espaços são superpostos se tem unidades em comum, e disjuntos caso contrário.

Uma área da memória é uma coleção não vazia de unidades elementares consecutivas. O endereço da área A , deno-

tado por $\text{end}(A)$, é o endereço da primeira de suas unidades, e o limite de A , $\text{lim}(A)$, é definido como sendo $\text{end}(A) + \text{tam}(A)$.

A área A precede a área B (e B segue A) se $\text{end}(B) \geq \text{lim}(A)$.

Se valer a igualdade, diremos que A segue imediatamente B . As áreas A_1, A_2, \dots, A_k são ditas consecutivas se A_i precede imediatamente A_{i+1} (para $1 \leq i < k$).

A memória é manipulada por meio de duas operações fundamentais: a consulta, que produz o estado corrente de uma unidade elementar, e a atribuição, que faz assumir um estado fornecido. Em ambas a unidade é especificada pelo seu endereço.

Essas operações podem ser estendidas a áreas de memória. Assim, dados o endereço e o tamanho de uma área, a consulta fornece seu conteúdo, e a atribuição permite substituí-lo por outro dado.

Neste trabalho não consideraremos memórias cuja estrutura não possa ser enquadrada na definição acima, como, por exemplo, memórias associativas e memórias de registro permanente ("write-once").

Quando nada for dito em contrário, estaremos supondo que o custo (ou tempo) para consulta ou atribuição de qualquer unidade elementar é constante, independentemente de seu endereço ou conteúdo. Suporemos também que a manipulação de uma área tem custo grosseiramente proporcional ao seu tamanho. Isso exclui, por exemplo, memórias de acesso sequencial (fitas e registradores de deslocamento).

1.3 Memória Paginada

A exceção mais importante da hipótese de custo uni-

forme na manipulação da memória são os sistemas de memória paginada ou virtual. Neles, a memória linear, onde as informações são armazenadas, é formada por dispositivos como discos e tambores magnéticos, cujo custo de consulta e armazenamento é elevado, mas praticamente independente do tamanho da área manipulada.

Para aumentar a eficiência dessa memória, ela é dividida em áreas relativamente grandes, chamadas páginas. Quando uma unidade elementar tiver que ser manipulada, toda a página que a contém é copiada para uma memória menor, com custo de acesso reduzido e uniforme, e a manipulação é feita sobre essa cópia.

Se já existir uma cópia da página em questão na memória rápida, a manipulação pode ser efetuada imediatamente evitando-se o acesso à memória lenta. Se tal não for o caso, e se a memória rápida estiver totalmente ocupada com cópias de páginas, uma ou mais destas são re-copiadas na memória lenta, abrindo espaço para a página desejada.

A eficiência deste esquema explica-se pela tendência que operações de consulta e atribuição tem de se concentrar em endereços próximos entre si. Isso eleva bastante a probabilidade de uma consulta ou atribuição usar unidades que estão nas mesmas páginas das usadas pela operação anterior, e que portanto já estão na memória rápida.

1.4 Registros e Campos

Uma implementação de uma linguagem num computador específico consiste de algoritmos que traduzem cada programa fonte escrito nessa linguagem para um programa objeto "equivalente", na linguagem de máquina desse computador.

Esses algoritmos devem estabelecer uma correspondência entre as variáveis do programa fonte e espaços da memória do computador. Os valores dessas variáveis devem corresponder aos conteúdos dos espaços. Comandos e formulas que manipulam essas variáveis e valores, no programa fonte, devem ser traduzidos para instruções da máquina que manipulam esses espaços e seus conteúdos.

O registro de uma variável, num dado instante, é o espaço de memória que estiver correntemente associado a ela. Os componentes de um registro são as partes do mesmo correspondentes aos componentes da variável a ele associada.

Neste trabalho admitiremos que a duas variáveis independentes correspondem sempre registros disjuntos. Para simplificar a descrição dos algoritmos, nos capítulos que seguem, suporemos também que cada registro é formado por uma única área de memória.

Esta última hipótese nem sempre vale na prática. Por exemplo, um trecho de matriz (slice) do ALGOL 68, em qualquer implementação razoável, pode corresponder a uma coleção de áreas não adjacentes, intercaladas eventualmente com áreas associadas a outros slices da mesma matriz.

Muitos dos resultados aqui expostos podem ser aplicados, com poucas modificações, mesmo aos casos em que essa hipótese não vale. Por razões de espaço não nos preocuparemos em indicar, em cada caso específico e em cada algoritmo, quais seriam essas modificações.

Um registro correspondente a uma variável simples é dito um campo. O registro de uma variável composta pode conter, além dos campos associados a seus componentes simples, uma ou mais áreas auxiliares, invisíveis ao programador. Os

exemplos mais comuns são enchimentos, usados para ajustar en direções e tamanhos de campos de acordo com as idiossincrasias da máquina; marcas, descritores ou indicadores, usados pelas rotinas e instruções do programa objeto para determinar as propriedades ou o estado do registro.

Suporemos essas áreas auxiliares disjuntas duas a duas, e completando com os campos todo o espaço ocupado pelo registro. Uns e outros serão chamados de itens do registro.

Embora uma variável simples seja indivisível para o programa fonte, pode ocorrer que apenas uma parte do campo associado seja usado para conter a codificação de seu valor, e o restante da área usado por enchimentos, marcas, descritores e indicadores referentes ao campo.

O formato de um registro é a especificação de seu tamanho, do número de itens que ele contém, e do tamanho e descrição de cada um.

Apresentaremos graficamente registros e seus itens como ilustrado na fig. Fl.1. A é um registro formado pelos itens A1, A2 e A3; B é outro registro, superposto a A e formado pelos itens A2, A3, A4 e A5. A figura indica também que A1, A2, A3, A4 e A5 são áreas consecutivas da memória.

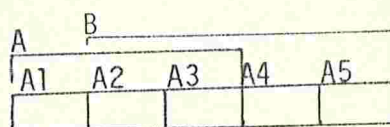


Figura Fl.1 - Representação de registros e seus itens.

1.5 Gerência de Memória

A associação de espaços de memória às variáveis do programa (a gerência da memória) pode ser feita de várias maneiras.

A técnica mais simples é a associação estática: a cada variável que o programa pode vir a usar, é associado um registro distinto, em tempo de compilação.

A alocação estática é muito ineficiente no uso da memória, e só é possível nas linguagens mais rudimentares. Nas linguagens como o ALGOL 60 e seus derivados é impossível, durante a compilação do programa fonte, estabelecer um limite para o número de variáveis usadas pelo mesmo.

Assim, hoje em dia usa-se universalmente a associação dinâmica: um espaço de memória é associado a uma variável só se, e quando, esta é criada; e a associação termina quando esta é destruída.

O espaço que, num dado momento, não está associado a nenhuma variável, e nem reservado para outros fins (p. ex., instruções), é chamado de espaço livre. Quando o programa especifica a criação de uma nova variável, o procedimento alocador escolhe uma área de tamanho adequado no espaço livre, e a associa à variável. A destruição de uma variável é realizada por um procedimento recuperador, que recolhe ao espaço livre o registro da mesma.

Nas linguagens com estrutura de blocos "pura", como no ALGOL 60, as variáveis destruídas são sempre as mais recentemente criadas das que existem no momento. Esse fato sugere a implementação do alocador e do recuperador baseados numa estrutura de pilha: o espaço livre é sempre uma única área, e o alocador extrai os registros sempre da mesma extre

midade desta. Quando um registro tiver que ser recolhido, ele será adjacente ao espaço livre, e portanto os dois poderão ser agregados numa única área.

A disciplina de pilha para criação e destruição de variáveis é excessivamente rígida para muitas aplicações. Em seu lugar, ou em paralelo a ela, as linguagens mais recentes possuem uma disciplina mais liberal: o programa pode criar novas variáveis independentemente da estrutura de blocos, e estas continuam existindo enquanto forem úteis ao programa.

Devido à ordem aleatória em que os registros são criados e destruídos, em sistemas com alocação liberal, o espaço livre pode ficar dividido em inúmeras áreas não adjacentes, o que complica a tarefa do alocador.

Em alguns casos, pode ser necessário relocar um registro; isto é, mover seu conteúdo para uma nova área no espaço livre, recolher a antiga, e atualizar o que for necessário no estado da computação para que a respectiva variável passe a ser associada à nova posição. A relocação pode ser usada para compactar todos ou alguns registros, aglomerando as áreas livres entre eles numa área única. Outras aplicações são a concentração dos registros num número pequeno de páginas de memória, ou a ampliação do tamanho de um registro após sua criação.

1.6 Referências e Apontadores

Um programa pode designar as variáveis a usar, em cada operação, de diversas maneiras. Uma variável pode ser especificada implicitamente, pelas regras da linguagem; como

resultante de uma operação de seleção ou indexação, aplicada a outra variável; ou por um valor especial (sua referência) que, seguida, designa univocamente a variável.

Referências podem ter várias formas, e podem ser obtidas de diversas maneiras, dependendo da linguagem. Podem constar explicitamente do programa, podem ser atribuídas e obtidas por consulta de variáveis, ou, em certos casos, podem resultar de operações envolvendo outros valores e de operações de leitura.

O acesso a variáveis por meio de referências complica bastante a gerência da memória. Se as operações permitidas pela linguagem puderem produzir a referência de qualquer variável, a recuperação liberal de espaço torna-se impossível: durante toda a execução, exceto talvez nos últimos instantes, é impossível garantir que uma variável nunca mais será usada pelo programa.

Por esse motivo, as linguagens que permitem a criação e destruição liberal de variáveis impõem restrições à maneira como as referências podem ser geradas.

A solução clássica para esse problema é fazer com que as únicas operações que podem criar um novo valor do tipo "referência" exijam como operando a variável designada, ou a criem na mesma ocasião. Veremos mais adiante como essas restrições permitem detetar algumas das variáveis que deixam de ser úteis, no decorrer da execução.

A representação interna de um valor de tipo referência é um apontador. Referências podem ser armazenadas em variáveis (simples, de tipo "referência"); um item de um registro que contém um apontador é dito um item apontador. Itens que não são apontadores são ditos atômicos. Um apontador de-

signa, ou aponta o registro da variável designada pela referência que ele codifica. Um registro apontado por alguém é dito um nó.

Apontadores são representados como na figura Fl.2. A, B e C são nós com itens $[A1, A2, A3]$, $[B1, B2]$ e $[B2, C2, C3, C4]$, respectivamente (os nós B e C são superpostos). O item A2 aponta para o nó B, A3 aponta para C, B1 aponta para A, e C4 aponta para o próprio C. Os demais itens são atômicos.

A figura Fl.2 indica também que A1, A2 e A3 são ad jacentes e ocorrem nessa ordem dentro da memória; o mesmo vale para B1, B2, C2, C3 e C4. Nada se pode afirmar, entretanto, a respeito das posições relativas das áreas A e B∪C, exceto que elas são disjuntas.

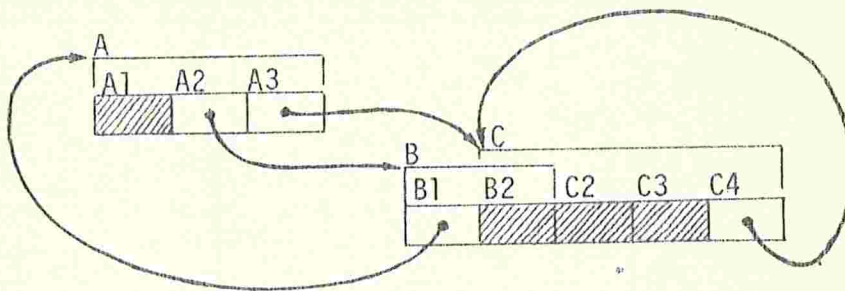


Figura Fl.2 - Representação de nós e apontadores.

1.7 Variáveis Ativas e Inativas

Num momento qualquer durante a execução, algumas das variáveis podem ser potencialmente acessíveis ao programa por

serem variáveis internas, temporários, variáveis de manipulação implícita, etc.; por serem mencionadas explicitamente nas instruções; ou ainda por serem componentes dessas. Em algumas linguagens, pode haver uma classe especial de variáveis, cujas referências podem ser geradas sem obedecer às restrições expostas acima (esse é o caso dos "átomos" na linguagem LISP 1.5, por exemplo, cujo nome pode ser lido ou gerado em tempo de execução).

Variáveis com as propriedades acima são ditas variáveis de base, e as demais são ditas flutuantes. O mesmo vale para os correspondentes registros.

Uma cadeia de variáveis é uma sequência de uma ou mais variáveis, onde a primeira é de base, e cada uma das outras é componente da anterior, ou designada pelo valor desta. O término da cadeia é a última dessas variáveis.

Seja A o conjunto das variáveis que, em determinado instante, são termos de cadeias de variáveis, e I o das demais variáveis. Note-se que A contém todas as variáveis de base, e que uma variável em A não é componente e nem designada pelo valor de nenhuma variável em A (senão, a cadeia que termina nesta poderia ser estendida até a primeira).

Suponha por absurdo que, em algum instante posterior, o programa manipule alguma variável de I , e considere a primeira ocasião em que isso ocorre. Essa operação não pode se referir explicitamente à variável ou ao seu nome, pois ela não é de base; não pode obtê-la por seleção a partir de outra, pois esta outra teria que estar em A ; deve portanto obtê-la seguindo uma referência. Mas esta referência não pode ser extraída de uma variável (que seria de A) e nem construída sem primeiro obter a variável que ela designa. Concluímos portanto que as variáveis em I serão inativas até o fim

do processamento. Apenas as variáveis de A são (potencialmente) ativas. As mesmas denominações serão usadas para seus registros.

Ao recuperador compete a tarefa de detectar as variáveis que se tornaram inativas e recolher os registros destas. Como as cadeias de variáveis podem ser criadas, destruídas e modificadas pelo programa de maneira imprevisível, esta tarefa é bastante complexa e trabalhosa, e tem que ser contínua, ou repetida periodicamente.

A gerência das variáveis de base, que geralmente obedecem à disciplina de pilha, é relativamente simples, e não será tratada neste trabalho.

A relocação de registros também é complicada pela presença de referências: quando uma variável é associada a um novo registro, todos os campos apontadores que designavam o registro antigo, ou componentes deste, devem ser localizados e alterados, de modo a apontar para a nova área.

O espaço livre e os registros flutuantes constituem juntos o espaço de alocação, e os registros de base formam o espaço de base.

Note-se que a cadeia de variáveis mais curta que termina numa variável dada inclui apenas uma variável de base, a primeira da cadeia. Portanto, referências que designem variáveis de base podem ser ignoradas na determinação das variáveis ativas.

Como além disso não vamos tratar da gerência dos registros de base, adotaremos a hipótese de que todos os apontadores designam nós flutuantes; os que apontariam para nós de base serão considerados átomos. Em consequência, "nó" será sinônimo de "nó flutuante".

1.8 Cadeias, Estruturas e Árvores

Uma cadeia de apontadores (ou simplesmente cadeia) de comprimento m é uma sequência alternada de m itens apontadores e m nós $(A_1, X_1, A_2, X_2, \dots, A_m, X_m)$ onde $m > 0$, cada A_i é um item do nó X_{i-1} (para $1 < i \leq m$), e designa o nó X_i (para $1 \leq i \leq m$). O item A_1 é o início da cadeia, e X_m término. Se A_1 for um item de X_m , a cadeia é fechada. Duas cadeias são paralelas se forem distintas e tiverem mesmo início e mesmo término. Se todos os itens forem distintos, a cadeia é uma trilha; se todos os nós o forem, ela é um caminho.

Os descendentes imediatos (ou filhos) de um registro são todos os nós apontados pelos itens não atômicos do registro, que por sua vez é um ascendente imediato (ou pai) de cada um desses nós.

Um nó X diz-se atingível ou acessível a partir de um item A se existir uma cadeia com início A e término X . Os descendentes de um item (ou registro) são todos os nós atingíveis a partir do mesmo (ou dos itens do mesmo), e ele é um ascendente desses nós. Note-se que um nó é descendente de si próprio se e somente se existir uma cadeia fechada na qual ele ocorre.

Note-se que um registro é ativo se e só se for um registro de base, ou for um nó atingível a partir de um campo de base, ou for um componente de um nó nestas condições.

A estrutura apontada por um item é a união de mesmo com todos os registros atingíveis a partir dele. Se o item for atômico, diremos que a estrutura apontada pelo mesmo é atômica.

Cada item de um nó aponta para uma estrutura, eventualmente atômica; essas são as sub-estruturas do nó.

A estrutura apontada por um item A é uma árvore se cada nó da mesma tiver sub-estruturas duas a duas disjuntas.

Isso equivale a dizer que para cada nó da estrutura existe uma única cadeia com início em A e término nele; o comprimento dessa cadeia é a profundidade do nó. Uma floresta é uma coleção de árvores disjuntas.

A profundidade da árvore é zero se ela for atômica ; caso contrário é a maior das profundidades de seus nós. Os nós da árvore que estão à mesma profundidade k constituem o nível k da árvore. O único nó de profundidade 1, se existir, é a raiz, e os que não tem descendentes são as folhas da árvore. A largura da árvore é a cardinalidade do nível com mais nós.

Uma lista ligada é uma estrutura na qual cada nó tem no máximo um filho. A lista é linear se não contém cadeias fechadas, e circular se existe uma cadeia fechada que contém todos os nós.

1.9 Codificação do Tipo de Nós e Campos

Procuraremos ignorar, na medida do possível, os detalhes da codificação dos valores da linguagem, e seu armazenamento nos registros da memória. Entretanto, teremos que nos preocupar com a maneira como o tipo dos nós e de seus campos são codificados.

Dizemos que a implementação utiliza nós auto-descritos se o tipo e formato de cada nó, incluindo o número de seus campos e os tipos destes, forem codificados por um ou mais descritores, localizados em posições fixas dentro do próprio nó.

O tamanho dos descritores não precisa ser muito gran

de. Um dado programa usa sempre um número finito de tipos distintos de variáveis, de modo que um descriptor pode se reduzir a apenas um inteiro (não muito grande) que identifica um desses tipos. Uma tabela separada pode conter o número de campos, e a descrição destes, para cada tipo de nó.

Se os nós são auto-descritos, os apontadores podem consistir apenas do endereço do registro apontado.

Uma vantagem importante das implementações com nós auto-descritos é que nelas é possível o exame sequencial de dois ou mais nós adjacentes. De fato, dado o endereço de um nó, seus descritores permitem determinar seu tamanho, e portanto o endereço do registro imediatamente seguinte a ele na memória.

Quando o tipo de cada nó não está completamente codificado no próprio nó, diz-se que a implementação usa nós amorfos. As linguagens de alto nível que permitem variáveis sobrepostas, como por exemplo, o ALGOL 68, exigem uma implementação desta espécie. Nestas linguagens, pode haver inúmeros nós, de tipos e tamanhos diferentes, começando no mesmo endereço da memória; portanto, não há como colocar em cada um seu próprio descriptor.

Mesmo em linguagens que permitem variáveis sobrepostas, as variáveis simples (e portanto os campos) nunca o são. Portanto, é sempre possível incluir em cada campo um descriptor que codifique seu tipo e formato, obtendo-se campos auto-descritos.

Em implementações com campos auto-descritos, um apontador precisa conter apenas informações sobre o endereço e o número de campos do nó designado. Além disso, o percurso sequencial da memória ainda é possível, campo-a-campo em vez

de no-a-no.

Na linguagem ALGOL 68, e em outras similares, os campos podem ser bastante pequenos, reduzindo-se até mesmo a um único "bit". A inclusão de um descritor em cada campo pode ser nesses casos muito dispendiosa; o que torna aconselháveis, para essas linguagens, implementações com nos e campos amorfos.

Em implementações com campos amorfos cada apontador deve conter, além do endereço do no apontado, o número de campos e a descrição dos tipos dos mesmos. Felizmente, no ALGOL 68 (e em muitas linguagens semelhantes), boa parte desas informações é conhecida em tempo de compilação. Apenas o número de componentes de nos de modo row, e o indicador de sub-tipo dos objetos de tipo union, precisam ser incluídos nos apontadores; as demais informações, sobre o no designado são dadas pelo tipo do apontador. Se este apontador corresponder a uma variável declarada, seu tipo é conhecido pelo compilador; se tiver sido obtido seguindo-se outro apontador, o tipo deste permite determinar o tipo do primeiro, e assim por diante.

Podemos ainda distinguir entre implementações com nos uniformes (isto é, que obrigam todos os nos a terem o mesmo tamanho) e as com nos de tamanho variado. Ou, ainda , entre as com campos uniformes e as com campos de tamanho variado.

A linguagem LISP 1.5 é o exemplo mais importante dos sistemas com nos e campos uniformes. Muitas implementações experimentais de linguagens de alto nível, e muitos programas de aplicação, usam nos de tamanho variado e campos de tamanho fixo, correspondendo estes últimos em geral às "pa-lavras" da memória do computador. Finalmente, nos e campos

de tamanhos variados são quase que obrigatórios em implementações eficientes das linguagens como ALGOL 68 e PASCAL.

1.10 Classificação dos Recuperadores

Podemos dividir os recuperadores de espaço livre em duas grandes categorias: os métodos de marcação e coleta, e os de recuperação por cópia.

Os primeiros consistem de uma etapa de marcação, que seguindo todas as cadeias emanantes dos campos de base, identifica todos os nós ativos; e de uma etapa de coleta, que recolhe ao espaço livre todas as áreas que não fazem parte dos nós encontrados na marcação.

Os recuperadores por cópia utilizam um espaço auxiliar, do mesmo tamanho do espaço de alocação, e, seguindo todas as cadeias que partem dos campos de base, relocam os nós ativos assim encontrados para posições consecutivas dentro desse espaço auxiliar. Ao fim dessa operação, o espaço auxiliar passa a ser usado como espaço de alocação; e este é ignorado até a próxima recuperação, quando será usado como auxiliar.

Ao fim da recuperação por cópia, o espaço livre é formado por uma única área. Nos métodos de marcação e coleta, pelo contrário, o espaço livre é obtido na forma de um grande número de áreas intercaladas com nós acessíveis. Em muitos casos, essa característica pode obrigar ao uso de um procedimento de compactação, quer independente, quer combinado com a etapa de coleta.

Outra maneira de classificar os recuperadores é baseada na maneira como sua execução é distribuída. Um recuperador serial é invocado várias vezes ao longo do processamen

to, por exemplo quando o alocador não consegue encontrar espaço livre suficiente, ou quando o movimento de páginas da memória virtual se tornou excessivo. Durante a execução de um recuperador serial, o processamento do programa do usuário é suspenso, e só é retomado quando a recuperação de todo o espaço de alocação estiver completa.

Em contraste, a execução de certos recuperadores pode ocorrer simultaneamente com a do programa usuário, ou, pelo menos, alternando-se a este em intervalos muito menores que o tempo total da recuperação. O objetivo desta técnica é evitar a interrupção do programa usuário por períodos demasiadamente longos, o que justifica a denominação de recuperadores em tempo real dada a tais algoritmos.

CAPÍTULO II - A REPRESENTAÇÃO DOS ALGORITMOS

2.1 Generalidades

A notação usada neste trabalho para representar os algoritmos assemelha-se a uma linguagem de programação. Ela não pretende estar completa e rigorosamente definida, e nem ser facilmente implementada e eficientemente compilável. Acreditamos que ela seja razoavelmente auto-explicativa, de modo que sua descrição se limitará aos aspectos essenciais, e, mesmo assim, apenas aqueles que serão usados neste trabalho.

A notação imita o estilo e a estrutura das linguagens de programação da família do ALGOL. Sua semântica entretanto, é mais próxima às das linguagens de implementação de sistemas, como BLISS {Wul/Rus/Hab 12/71} e BCPL {Ric 69}. É importante notar que a linguagem não possui alocação dinâmica de variáveis, sistema rígido de tipos, e outras características sofisticadas de linguagens de alto nível, pois ela se destina a descrever a implementação dessas mesmas características. Em particular, admitiremos a manipulação de endereços por operações aritméticas.

2.2 Condicionais, Repetições e Blocos

As construções `se α então β senão γ , e repete, enquanto α , β , equivalem respectivamente às construções if α then β else γ fi e while α do β od da linguagem ALGOL 68.`

Blocos (comandos compostos) serão indicados por colchetes

tes. Por exemplo, a sequência `begin α ; β ; γ end` seria escrita como abaixo:

$$\left[\begin{array}{l} \alpha; \\ \beta; \\ \gamma; \end{array} \right.$$

A construção `repete α até que β` equivale em ALGOL 68 `begin α ; while β do α od end.` O correspondente ao comando iterativo `for v from i step σ until ψ do α od` seria `repete`, para v desde i passo σ até ψ α . Como em ALGOL 68, suporemos que v é local ao comando iterativo; passo σ pode ser omitido se $\sigma = 1$. O significado de `repete` indefinidamente α é `while true do α od`.

A instrução `termina ρ` , se executada dentro de um comando rotulado ρ , causa o término imediato deste, equivalente a um `goto` para o comando seguinte ao mesmo.

A principal finalidade do comando `termina` é encerrar malhas que não podem ser escritas diretamente com `repete`, enquanto... Veja-se o exemplo abaixo.

MALHA: `repete indefinidamente`

$$\left[\begin{array}{l} \alpha; \\ \text{se } \beta \text{ então termina MALHA;} \\ \gamma; \end{array} \right.$$

que, em ALGOL 68, seria escrito

`begin`

`do α ; if β then goto FIM fi; γ od;`

`FIM: end`

2.3. Áreas, Modos e Declarações

Os objetos manipuláveis na nossa notação são áreas e seus conteúdos (dados). Um dado pode representar um inteiro,

valor lógico, um endereço, e assim por diante, oportunamente codificados.

O comando de atribuição tem a forma $\alpha \leftarrow \beta$, onde α é uma fórmula que especifica uma área, e β outra fórmula que determina um dado. Seu efeito é armazenar o dado β na área α .

Ocorre frequentemente que a informação codificada pelo dado β deve ser recodificada, segundo outras convenções, antes de ser armazenada em α . Para não ter que especificar explicitamente essas informações, introduziremos uma série de regras que, para cada fórmula, permitem determinar o modo de uso da área (ou dado) por ela denotada.

O modo de um dado informa como o mesmo deve ser interpretado (inteiro, endereço, etc.), como foi codificado, e qual o seu tamanho. O modo de uma área descreve os conteúdos que nela podem ser armazenados. Numa atribuição $\alpha \leftarrow \beta$ o dado β é implicitamente recodificado, se necessário, segundo o modo de α , mantendo sua interpretação.

As áreas usadas pelos algoritmos serão declaradas no início dos blocos em que elas forem necessárias. Apesar da semelhança dessa convenção com as regras de escopo do ALGOL 60, não estamos supondo que a alocação das áreas seja dinâmica.

As declarações tem a forma área $\alpha : \mu$ onde α é o identificador da área e μ seu modo. O modo pode ser re-especificado por ocasião do uso; a construção α como v especifica uma área com mesmo endereço de α mas com modo de uso e tamanho dados por v . O modo e o tamanho especificados na declaração de α são ignorados neste caso.

A especificação de uma área, num contexto onde seria necessário um dado, estará denotando o conteúdo corrente da mesma.

2.4 Modos Inteiro, Booleano e Dado

O modo inteiro de τ_1 a τ_2 especifica que o dado por ele qualificado deve ser entendido como a codificação de um número inteiro, no intervalo de τ_1 a τ_2 (inclusive).

O tamanho de tal dado e os detalhes da codificação serão deixados em aberto. Suporemos apenas que, em cada implementação, ambos são univocamente determinados pelo intervalo $\tau_1 - \tau_2$.

Uma regra de codificação que poderia corresponder a esse modo, numa implementação típica, representaria o inteiro k por uma sequência de "bits" de tamanho $\lceil \log_2(\tau_2 - \tau_1 + 1) \rceil$, correspondente à codificação binária de $k - \tau_1$.

Se os modos de α e β forem inteiro de τ_1 a τ_2 e inteiro de τ_3 a τ_4 , respectivamente, a atribuição $\alpha + \beta$ atribui a α um conteúdo que, interpretado segundo o modo de α , corresponde ao mesmo valor numérico de β , interpretado segundo o modo deste.

Fórmulas denotando dados de modos inteiros podem ser combinadas por operações aritméticas. As operações são efetuadas sobre os valores numéricos representados por esses dados. O resultado de cada operação é codificado segundo um modo inteiro de τ_1 a τ_2 , com τ_1 e τ_2 suficientes para abranger todos os resultados possíveis da mesma.

Um dado de modo booleano é a codificação (novamente, segundo regras dependentes da implementação) de um dos valores lógicos, 'verdadeiro' ou 'falso'.

Um dado pode ter modo dado de tamanho τ ; nesse caso, deve ser interpretado como ele próprio, sem codificação nenhuma. Este modo será usado sempre que o conteúdo de uma área for irrelevante para o algoritmo, importando apenas seu

tamanho. A atribuição $\alpha + \beta$, onde α e β tem modo dado, só será usada quando os dois tiverem o mesmo tamanho.

2.5 Modo Endereço

Um dado β de modo endereço será interpretado como codificação do endereço de uma unidade de memória. Suporemos que o tamanho de β é constante em cada implementação.

Se ϵ é um endereço, a expressão $\epsilon \uparrow$ como μ denota a área de endereço ϵ cujo tamanho e modo são os dados por μ . Se β for um dado de modo endereço de μ , β é um endereço; a expressão $\beta \uparrow$ equivale a $\beta \uparrow$ como μ , e denota a área de modo μ cujo endereço codificado é β .

Em operações aritméticas, os endereços são interpretados como inteiros. Por exemplo, $\epsilon + 1$ é o endereço da unidade elementar de memória imediatamente seguinte à de endereço ϵ , e $\text{end}(\alpha) + \text{tam}(\alpha)$ é o da área imediatamente seguinte à área α .

2.6 União de Modos

Em certos contextos, uma mesma fórmula pode resultar em áreas ou dado de modos distintos, cada vez que for calculada. Para descrever essa situação, usaremos eventualmente um modo da forma união de $(\mu_1, \mu_2, \dots, \mu_n)$, onde $\mu_1, \mu_2, \dots, \mu_n$ são os modos possíveis do resultado da fórmula.

A interpretação e o tamanho de um dado β com tal modo são indefinidas; sempre que for necessário usá-lo, indicaremos explicitamente, pela construção β como μ_i , qual das interpretações deve ser considerada.

Um dado ou área α com modo união disjunta de $(\mu_1, \mu_2, \dots, \mu_n)$ é a codificação de um inteiro i entre 1 e n (o prefixo de α), seguida de um dado ou área de modo μ_i (o cerne de α).

A conversão de um dado β de modo μ_i para o modo união disjunta de $(\mu_1, \mu_2, \dots, \mu_n)$ será subentendida sempre que necessário. A conversão contrária, isto é, a obtenção do cerne de um dado com este modo, será indicada pela construção $\text{cerne}(\beta)$, ou implicitamente se necessário. Para áreas, só é aplicável a conversão neste sentido.

2.7 União Uniforme

As construções união e união disjunta permitem construir um modo que descreve dados ou áreas de tamanhos variados. Em alguns casos, o tamanho de um desses objetos pode ser determinado, no momento da execução, a partir do próprio; é o caso, por exemplo, do modo união disjunta de (booleano; inteiro de 1 a 10). Para outros modos, nem isso é possível; considere-se o exemplo acima, com união em vez de união disjunta.

Somente usaremos a atribuição $\alpha + \beta$ quando β , após as conversões implícitas do caso, tiver mesmo modo que α , e os tamanhos de ambos puderem ser determinados (no mínimo, durante a execução) e forem iguais nesse momento. Esta regra ex-

clui, portanto, atribuições a áreas com modo união de(...), e as restringe bastante quando este modo é união disjunta de(...).

Para aliviar esse problema, usaremos eventualmente o modo união uniforme de $(\mu_1, \mu_2, \dots, \mu_n)$ que equivale a união de $(\mu_1, \mu_2, \dots, \mu_n)$, exceto que dados e áreas desse modo são completados com um enchimento adequado, de modo a terem todos o mesmo tamanho (que depende apenas de $\mu_1, \mu_2, \dots, \mu_n$, e da implementação considerada).

Analogamente, usaremos união uniforme disjunta de $(\mu_1, \mu_2, \dots, \mu_n)$, para descrever dados ou áreas de modo união disjunta de $(\mu_1, \mu_2, \dots, \mu_n)$, completados com enchimentos como descrito acima.

Um dado de modo μ_i , cujo tamanho seja determinável, pode ser convertido implicitamente para os modos união uniforme de $(\mu_1, \mu_2, \dots, \mu_n)$, ou união uniforme disjunta de $(\mu_1, \mu_2, \dots, \mu_n)$, pela anexação de prefixos e enchimentos adequados.

A construção $\text{cérne}(\alpha)$ pode ser aplicada a um dado ou área α de modo união uniforme disjunta, com o significado óbvio, e será subentendida se necessário.

A conversão de um dado ou área α , com modo união uniforme de $(\mu_1, \mu_2, \dots, \mu_n)$, para qualquer dos modos μ_i , deve ser indicada explicitamente pela construção α como μ_i .

Áreas de trabalho declaradas nos algoritmos devem ser reservadas com tamanho suficiente para o maior dos dados admitidos pelo modo usado na declaração.

2.8 Modos Compostos e Sequências

Se um dado ou uma área β for descrita por um modo da forma $[\alpha_1:\mu_1, \alpha_2:\mu_2, \dots, \alpha_n:\mu_n]$, o mesmo deve ser interpretado como uma sequência de n dados ou áreas consecutivas, de modos $\mu_1, \mu_2, \dots, \mu_n$.

Cada α_i é um identificador, que é usado na construção $\beta.\alpha_i$ para denotar o i -ésimo desses elementos, com modo μ_i .

Para que essa notação seja efetiva, só a usaremos se os tamanhos de todos os elementos de β que precedem $\beta.\alpha_i$ forem determináveis em tempo de execução.

O modo sequência de v μ é usado para descrever dados ou áreas formados pela concatenação de v elementos de modo μ . Se β é um objeto desse modo, e i é um inteiro entre 1 e v , a expressão $\beta[i]$ denota o i -ésimo dos elementos de β . Novamente, essa notação só será usada se o modo μ especificar os tamanhos dos elementos, ou se os mesmos puderem ser determinados durante a execução.

Se o modo de um dado ou área β for $[\alpha_1:\mu_1, \alpha_2:\mu_2, \dots, \alpha_n:\mu_n]$, e um dos μ_i mencionar um α_j (com $j < i$), este deve ser lido como denotando $\beta.\alpha_j$. Por exemplo, um dado de modo $[t: \text{inteiro de 1 a 10}, b: \text{dado de tamanho } t]$ consiste da codificação de um inteiro, seguida de um dado cujo tamanho depende desse inteiro.

Se $\beta_1, \beta_2, \dots, \beta_n$ forem dados de modos $\mu_1, \mu_2, \dots, \mu_n$, a expressão $[\alpha_1:\beta_1, \alpha_2:\beta_2, \dots, \alpha_n:\beta_n]$ denota o dado de modo $[\alpha_1:\mu_1, \alpha_2:\mu_2, \dots, \alpha_n:\mu_n]$ formado pela concatenação dos β_i na ordem dada.

A atribuição $\alpha \leftarrow \beta$, se α e β forem sequências com mesmo número de elementos, significa a atribuição de cada elemento de β ao correspondente de α .

2.9 Procedimentos

Procedimentos são definidos pela construção

procedimento π :

recebendo $\delta_1:\mu_1, \delta_2:\mu_2, \dots, \delta_n:\mu_n$,
e as áreas $\sigma_1:v_1, \sigma_2:v_2, \dots, \sigma_m:v_m$,
efetua θ

onde θ é um comando, $\delta_1, \delta_2, \dots, \delta_n$ são parâmetros passados "por valor", e $\sigma_1, \sigma_2, \dots, \sigma_m$ são os passados "por nome". A invocação de tal procedimento teria a forma

$$\pi(\beta_1, \beta_2, \dots, \beta_n, \alpha_1, \alpha_2, \dots, \alpha_m)$$

onde os β_i e α_j são fórmulas que produzem dados e áreas, respectivamente.

A chamada de tal procedimento corresponde à execução do comando

$$\left[\begin{array}{l} \text{áreas } \delta_1:\mu_1, \delta_2:\mu_2, \dots, \delta_n:\mu_n; \\ \delta_1 \leftarrow \beta_1; \delta_2 \leftarrow \beta_2; \dots; \delta_n \leftarrow \beta_n; \\ \theta' \end{array} \right.$$

onde θ' é obtido de θ substituindo-se toda ocorrência de σ_j por $(\alpha_j \text{ como } v_j)$. Estamos supondo, nesta explicação, que todos os identificadores declarados no algoritmo são distintos; é óbvio que, trocando-se adequadamente alguns identificadores, todo algoritmo pode ser reescrito nestas condições.

Note-se que uma atribuição a um dos σ_j altera o conteúdo da área α_j correspondente. Note-se também que pode haver recodificação dos dados β_i , se seu modo não coincidir exatamente com μ_i , mas que o modo e o tamanho das áreas α_j são ignorados.

2.10 Valores Simbólicos e Modos Degenerados

Os valores inteiros (0, 1, 2,...) e booleanos ('verdadeiro' e 'falso') são apenas casos particulares de valores simbólicos que tem um papel especial em certas operações e comandos. Podemos usar dados para representar muitos outros símbolos, além dessas duas classes, bastando estabelecer regras adequadas de codificação.

Nos algoritmos, tais valores simbólicos serão indicados por identificadores entre apóstrofes. Tais valores poderão ser armazenados em áreas de modos convenientes, comparados (com "=" ou "≠") e passados como parâmetros a procedimentos.

Para definir os modos de expressões e áreas que podem assumir valores simbólicos, teremos que começar definindo modos degenerados, que se restringem cada um a um único valor, e são escritos como este.

Assim, os modos 5 e 'falso' descrevem expressões cujos valores estão restritos, respectivamente, aos valores 5 e 'falso'. Se o modo de uma expressão é degenerado, o resultado da mesma pode ser codificado em zero bits. As constantes que ocorrem explicitamente nos algoritmos tem tais modos.

Modos degenerados são frequentemente usados em uniões disjuntas. Por exemplo, união disjunta de ('falso', 'verde', 5, 0) descreve um dado como sendo a codificação de um dos quatro valores mencionados. Se adotarmos, para os quatro modos degenerados, a codificação mais econômica (zero bits), tal dado reduz-se aos dois bits do prefixo da união.

Exceto pelo significado especial que a eles é atribuído, os modos inteiro de τ_1 a τ_2 e booleano podem ser considerados sinônimos de união uniforme disjunta de $(\tau_1, \tau_1+1,$

τ_1+2, \dots, τ_2) e união uniforme disjunta de ('verdadeiro', 'falso'), respectivamente.

2.11 Outras Características da Notação

Para evitar repetições desnecessárias, utilizaremos também declarações de modos. A declaração modo $\alpha:\mu$ permite que, no resto do bloco onde ela ocorre, o identificador α se ja usado para denotar o modo μ .

Recorreremos também, em alguns casos, à atribuição simultânea $(\alpha_1, \alpha_2, \dots, \alpha_n) \leftarrow (\beta_1, \beta_2, \dots, \beta_n)$, cujo efeito é o do bloco

$$\begin{array}{l} \alpha_1 \leftarrow \beta_1; \\ \alpha_2 \leftarrow \beta_2; \\ \vdots \\ \alpha_n \leftarrow \beta_n \end{array}$$

exceto que todas as expressões β_i são calculadas antes da primeira atribuição ser efetuada. Assim, $(\alpha, \beta) \leftarrow (\beta, \alpha)$ troca os conteúdos das áreas α e β entre si; esta operação será também denotada por $\alpha \leftrightarrow \beta$.

A instrução move β tamanho τ para α equivale à atribuição $(\alpha \text{ como dado de tamanho } \tau) \leftarrow (\beta \text{ como dado de tamanho } \tau)$. Se tamanho τ for omitido, deve-se entender tamanho $\text{tam}(\beta)$.

A instrução move β_1 tamanho τ_1 para α_1 , β_2 tamanho τ_2 para α_2, \dots, β_n tamanho τ_n para α_n , é definida, de modo análogo, como equivalente a uma atribuição simultânea.

Uma declaração da forma

área α : μ inicialmente β

especifica que uma atribuição implícita $\alpha \leftarrow \beta$ deve ser efetuada em algum momento, antes de todos os comandos que seguem.

Comentários explicativos, delimitados por chaves ($\{\}$), serão inseridos ocasionalmente nos algoritmos, sem nenhum efeito em sua execução.

A construção $\text{erro}\{\sigma\}$ será usada para indicar condições excepcionais detectadas num algoritmo, que o obriguem a encerrar prematuramente o processamento. O comentário $\{\sigma\}$ descreverá a condição detectada.

Usaremos livremente as flexões de gênero, número e tempo da língua portuguesa, nos termos e identificadores usados nos algoritmos, para melhorar a legibilidade destes.

Outras construções e operações, de uso mais restrito, serão definidas quando forem necessárias.

2.12 Análise da Eficiência dos Algoritmos

Para dar uma idéia da eficiência (em termos de tempo de execução) de cada método, colocaremos à extrema direita de cada comando, o número de vezes que este será executado em cada aplicação do algoritmo.

Esses números, e muitas das análises feitas no texto, dependem de vários parâmetros da implementação e do estado corrente do espaço de alocação. Os mais importantes são:

- M Tamanho total do espaço de alocação
- N Número total de nós flutuantes
- C Número total de campos flutuantes
- P Número total de campos apontadores em nós flutuantes
- Q Número total de campos atômicos desses nós ($=C-P$)

Os símbolos acima, com a afixação do índice "A", representam os parâmetros análogos, referentes apenas aos nós cativos. Com o índice "B", representam os mesmos parâmetros, referentes porém aos registros de base.

Por exemplo, M_A é o tamanho total dos nós ativos, C_A o número total de seus campos, P_B o número de apontadores de base, e assim por diante.

Usaremos também os parâmetros:

- \tilde{c} (\tilde{c}) Número máximo (médio) de campos por nó
- \tilde{m} (\tilde{m}) Tamanho máximo (médio) dos nós
- \tilde{p} (\tilde{p}) Número máximo (médio) de campos apontadores por nó
- \tilde{q} (\tilde{q}) Número máximo (médio) de campos atômicos por nó.

Suporemos que as médias \tilde{c} , \tilde{m} , \tilde{p} e \tilde{q} são as mesmas quando calculadas sobre todos os nós flutuantes e apenas sobre os ativos.

O tempo exigido para a execução dos algoritmos será estimado supondo-se que o tempo consumido numa única instrução elementar (comparação, atribuição ou cálculo aritmético)

é limitado por uma constante, ou por uma função linear do tamanho total dos dados envolvidos. Como as operações aritméticas mais complexas que usaremos são multiplicações e divisões por constantes pequenas, tal hipótese é justificável.

Diremos que uma quantidade τ relevante para a análise de um algoritmo (tempo, espaço auxiliar, número de operações, etc.) é no máximo $O(\epsilon)$, se, para cada implementação do mesmo, existirem constantes K e K' , tais que $\tau \leq K + K'\epsilon$, sempre que os parâmetros que ocorrem em ϵ sejam suficientemente grandes.

Diremos que τ é no mínimo (ou pelo menos) $O(\epsilon)$, se, para toda implementação do algoritmo, existirem constantes k e k' tais que, para toda combinação suficientemente grande dos parâmetros de ϵ , existir uma situação em que $\tau \geq k + k'\epsilon$.

Por exemplo, se $\tau = 3N + 6(C_B)^2 + 2$, podemos dizer que τ é no máximo $O(N + (C_B)^3 + P_A)$, pelo menos $O(\sqrt{N} + C_B)$, e que τ é $O(N + (C_B)^2)$.

2.13 Definições de Modos e Áreas de Uso Geral

A descrição formal dos modos $\bar{\text{átomo}}$, apontador, campo, etc., poderá variar de um algoritmo para outro, conforme as necessidades. Se não forem redefinidos, no texto ou nos algoritmos, estaremos adotando as seguintes definições para os mesmos:

modos campo: união disjunta de ($\bar{\text{átomo}}$, apontador),

apontador: endereço de $\bar{\text{nô}}$,

$\bar{\text{átomo}}$: união disjunta de $(\mu_1, \mu_2, \dots, \mu_n)$

onde $\mu_1, \mu_2, \dots, \mu_n$ são um ou mais modos, dependendo da instalação, que não incluem endereços de áreas internas ao espaço de alocação.

Admitiremos que este último é uma seqüência de N nós, e que as áreas INI e LIM , de modo endereço, contém seu endereço e seu limite, respectivamente. Suporemos que o espaço de base é uma seqüência de C_B campos, de nome R .

O modo $nó$ será definido, em cada caso, no texto ou nos algoritmos.

CAPÍTULO III - ALGORITMOS DE PERCURSO E MARCAÇÃO

3.1 Introdução

Neste capítulo, estudaremos os algoritmos mais importantes para o percurso e a marcação de estruturas. O percurso consiste em localizar todos os nós atingíveis a partir dos campos de base, seguindo todos os seus campos apontadores. A marcação consiste em armazenar em cada nó a informação "acessível" ou "inacessível", em itens especialmente reservados para esse fim (marcas).

Além de importantes em si próprios, os algoritmos de percurso e marcação interessam-nos por constituírem a primeira etapa de muitos recuperadores.

Essas duas operações são inseparáveis. Por um lado, é aparentemente impossível determinar se um nó é acessível ou não sem seguir todos os apontadores de todos os nós acessíveis a partir do espaço de base. Por outro, estruturas com ciclos e cadeias paralelas só podem ser percorridas com segurança e eficiência por algoritmos que sejam capazes de determinar se um nó foi visitado anteriormente, ou não; e a marcação dos nós visitados é a solução mais natural para este problema.

Trataremos inicialmente o caso de implementações que não permitem nós sobrepostos. Para tais implementações, podemos supor sem perda de generalidades que os nós são auto-descritos.

Sem nós sobrepostos, é suficiente ter uma única mar

ca, com dois estados possíveis, cada nó. Para maior simplicidade, suporemos ainda que a marca é um dos itens do nó, e ocupa uma posição fixa dentro do mesmo.

Nos algoritmos que se seguem, se não for explicitamente declarado de outra forma, estaremos considerando o modo nó como descrito abaixo:

```
modos  nó:  marca: marca-de-nó,
           c: inteiro de 1 a  $\hat{c}$ ,
           corpo: sequência de c campos,
           marca-de-nó: união uniforme disjunta
                     de ('útil', 'inútil')
```

Se os nós tiverem tamanho uniforme, as marcas podem ser agrupadas numa área separada, fora do espaço de alocação, na forma de uma área marcas: sequência de N marcas-de-nós. A marca do nó de endereço ϵ será, então, $\text{marca}[(\epsilon - \text{INI})/\hat{m}]$.

Como as marcas são usadas apenas durante a recuperação, sua alocação numa área separada pode resultar em economia de memória. Por exemplo, o espaço para elas pode ser providenciado transferindo-se temporariamente o programa objeto para um dispositivo auxiliar de armazenamento.

No início da recuperação, todas as marcas devem conter o valor 'inútil'. Quando um algoritmo de percurso e marcação determina que um nó é acessível, a marca deste recebe o valor 'útil', permanecendo assim até o fim do algoritmo. Um nó com $\text{marca} = \text{'útil'}$ será dito marcado.

3.2 Ordem de Marcação e Árvore de Percurso

Um algoritmo de marcação (para implementações com nós

disjuntos) define uma ordem de marcação sobre os nós percorridos, que é a ordem em que os mesmos são marcados como 'úteis',

Para marcar um nó, o algoritmo deve obter seu endereço a partir de um campo apontador, situado no espaço de base ou em outro nó. Os campos que foram realmente usados dessa forma pelo algoritmo são ditos primários, e os demais secundários.

Na maioria dos algoritmos que veremos neste capítulo, todo nó é marcado apenas uma vez, logo que é obtido pelo algoritmo. Se substituirmos idealmente todos os campos secundários por átomos, é fácil verificar que todo nó marcado continua atingível a partir de um campo de base por uma única cadeia. A estrutura apontada por cada campo de base, com essa transformação, é dita uma árvore de percurso, e a coleção dessas árvores é a floresta de percurso. Indicaremos por N_{A1} o número dessas árvores, que é também o número de raízes da floresta. O número de apontadores primários da floresta de percurso é portanto $N_A - N_{A1} = N_{A+}$ = número de nós que não são raízes da floresta de percurso.

A relação entre a floresta de percurso e a ordem de marcação permite dividir os algoritmos deste capítulo em duas grandes classes: horizontais e verticais.

3.3 Algoritmos Horizontais

Estes algoritmos caracterizam-se pelo fato que todos os descendentes diretos de um mesmo nó, na floresta de percurso, ocorrem em posições consecutivas na ordem de marcação.

O algoritmo A3.1 é um exemplo típico. Ele utiliza

procedimento MARCA-HORIZONTAL:

efetua

1	[área PE: fila de $N - \lceil (N - C_B)/\hat{p} \rceil$ apontadores inicialmente vazia;
		repete, para J desde 1 até C_B ,
C_B		[se $R[J]$ não é apontador, então
P_B		e se $R[J].\downarrow.marca$ é 'inútil', então
N_{A1}		[$R[J].\downarrow.marca \leftarrow 'útil'$;
		$PE \Leftarrow R[J]$;
		áreas E, A: apontadores;
		repete, enquanto PE não é vazia,
N_A		[$E \Leftarrow PE$;
		repete, para J desde 1 até $E.\downarrow.c$,
C_A	[se $E.\downarrow.corpo[I]$ é apontador, então	
	[$A \leftarrow E.\downarrow.corpo[I]$;	
P_A	se $A.\downarrow.marca$ é 'inútil', então	
N_{A+}	[$A.\downarrow.marca \leftarrow 'útil'$;	
	$PE \Leftarrow A$	

Algoritmo A3.1 - Marcação Horizontal de estruturas.

é retirado de PE, são inseridos no máximo \hat{p} outros. Por indução, verifica-se que depois de inserir k elementos, não haverá em PE mais que $N_{A1} + k - \lceil k/\hat{p} \rceil$. Como nenhum endereço de nó é inserido duas vezes em PE, temos $k \leq N_{A+}$, donde segue o limite acima enunciado.

Este limite pode ser atingido. Se PE for uma fila, o pior caso é dado por uma estrutura de "árvore completa", como a da figura F3.3A. A figura F3.3B ilustra o pior caso quando PE é uma pilha.

Para a maioria das estruturas, o número de entradas realmente necessárias em PE é sensivelmente menor que esse limite. Se PE for uma fila, esse número é exatamente a largura da árvore de percurso; se PE for pilha, esse número nunca é mais que $\hat{p}(\ell-1)+1$, onde ℓ é a profundidade da árvore.

Ora, sabe-se {Knu 68, } que a profundidade média de uma árvore aleatória com N_A nós é $O(\log N_A)$. Uma árvore com N_A nós e essa profundidade tem largura pelo menos $O(N_A/\log N_A)$, que cresce mais depressa que $O(\log N_A)$ quando N_A aumenta.

Isso parece justificar a utilização do algoritmo A3.1 com uma pilha PE (e não uma fila) de bem menos que $N_A - \lceil N_{A+}/\hat{p} \rceil$ entradas, recorrendo-se a unidades auxiliares de armazenamento (discos e tambores magnéticos) nos raros casos em que a mesma for insuficiente.

Infelizmente, uma análise rigorosa dos dois métodos parece ser muito difícil. Um fato que complica a análise, e tira um pouco do peso da conclusão acima, é que, numa estrutura genérica, a árvore de percurso induzida pelo algoritmo com pilha é diferente (e mais profunda, em geral) do que a obtida com fila.

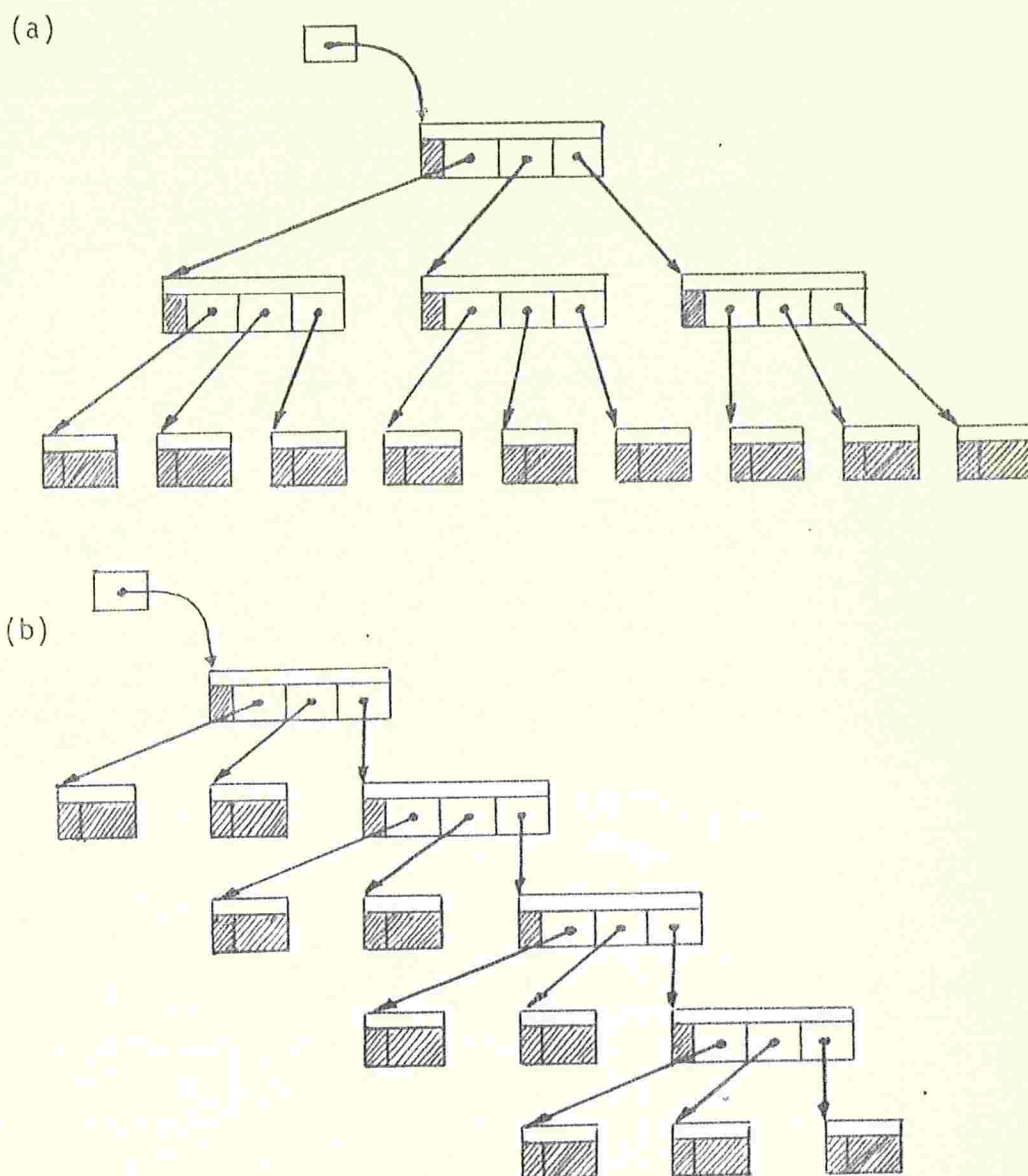


Figura F3.3 - Exemplos de estruturas com $N_A = 13$, $\hat{p} = 3$ que exigem $N_A - N_{A+}/\hat{p} = 9$ elementos em PE:
 (a) quando PE é fila, (b) quando PE é pilha.

Mesmo assim, o uso de uma pilha parece ser mais aconselhável. Se analisamos o outro caso, foi porque o mesmo ilustra melhor a "horizontalidade" do percurso, e por ter grande semelhança com certos recuperadores por cópia do capítulo V.

3.5 Otimização do Uso da Pilha

Pode-se indagar até que ponto a escolha adequada da ordem de inserção e retirada de elementos na área auxiliar PE pode otimizar o uso da mesma. Isto é, suponha que a operação $E \leftarrow PE$ do algoritmo A3.1, em vez de retirar o elemento mais recente ou mais antigo dos armazenados em PE, retira o escolhido por uma função (ou "oráculo") arbitrária. Será que existe então um oráculo "ótimo", que, na marcação de qualquer estrutura, minimiza o número máximo de entradas de PE contemporaneamente ocupados? E como é esse oráculo? Estamos supondo que o oráculo dispõe de quaisquer recursos necessários para tomar sua decisão, de modo que ele não precisa ser eficiente, nem mesmo computável.

O oráculo ótimo para árvores é facilmente descrito, embora não seja eficiente. Considere a função $S(\epsilon)$, definida, para cada endereço ϵ de nós da árvore, pela fórmula

$$S(\epsilon) = \max(\{1\} \cup \{k-i+S(\epsilon_i) \mid 1 \leq i \leq k\})$$

onde $\epsilon_1, \epsilon_2, \dots, \epsilon_k$ são os endereços dos filhos do nó ϵ , ordenados de modo que $S(\epsilon_1) \leq S(\epsilon_2) \leq \dots \leq S(\epsilon_k)$

O oráculo ótimo é o que escolhe, para ser retirado de PE, o endereço ϵ que tiver menor $S(\epsilon)$. Não é difícil verificar que, com tal oráculo, se inicialmente PE contém apenas o endereço ϵ , o número máximo de elementos simultaneamente em PE é exatamente $S(\epsilon)$. Verifica-se também que

nenhum outro oráculo exige menos espaço em PE.

Uma característica da função S é que se ϵ' é filho de ϵ , então $S(\epsilon') \leq S(\epsilon)$. Portanto, da segunda vez em diante, o oráculo descrito acima sempre retira de PE um dos descendentes do nó cujo endereço foi escolhido na vez anterior. Esse efeito pode ser conseguido inserindo-se sempre os endereços ϵ_i dos filhos de cada nó examinado em ordem decrescente de $S(\epsilon_i)$, e usando-se PE com disciplina de pilha.

A função $S(\epsilon)$ pode ser interpretada como a quantidade máxima de "ramificações colaterais" encontradas ao longo de uma cadeia que comece com ϵ . Essa função não pode ser calculada sem percorrer a estrutura apontada por ϵ , de modo que o oráculo ótimo acima é impraticável.

Além do mais, a determinação da escolha ótima para estruturas genéricas, com circuitos e cadeias paralelas, é um problema muito mais difícil. Não podemos sequer garantir que o tempo necessário para essa determinação seja limitado por um polinômio no número de nós da estrutura.

Porém, a análise acima oferece algumas heurísticas que diminuem em média o número de entradas necessárias em PE. Por exemplo, se os nós da estrutura apontada por um endereço ϵ em PE tiverem todos no máximo um campo apontador, então ϵ é uma escolha ótima em qualquer caso. Tal condição pode ser determinada, às vezes, pelo tipo do nó $\epsilon+$; por exemplo, em ALGOL 68, objetos com o "modo".

```
mode list = (int a, ref list b)
```

tem essa propriedade.

Outro exemplo é a linguagem LISP 1.5 [Cla/Gre 2/77]. Devido à notação usada para a representação das S-expressões,

o campo CAR de cada nó geralmente aponta para uma estrutura menor e mais simples que a apontada pelo CDR. Portanto, um uso mais eficiente da pilha PE pode ser obtido retirando-se sempre campo CAR de cada nó antes do campo CDR, ou seja, empilhando-se este antes daquele.

3.6 Otimização do Número de Falhas de Páginas

Em implementações com memória virtual, a economia de espaço geralmente é um aspecto secundário dos algoritmos. Muito mais importante do que isso é o número de falhas de páginas, isto é, o número de vezes que o algoritmo tenta consultar ou alterar uma página que não está na memória rápida. Tais falhas causam a transferência da página em questão para a memória rápida, e eventualmente a de outra página para a memória lenta; são portanto operações consideravelmente demoradas.

Para estimar o número de falhas de páginas (ou número de falhas, apenas) faremos algumas hipóteses simplificadoras.

Em primeiro lugar, suporemos que o espaço de base, o algoritmo, e as áreas de trabalho usadas por ele estão permanentemente na memória rápida. O espaço de alocação é dividido em G páginas, e cada nó está integralmente contido numa única página.

Admitiremos que os nós ativos estão aleatoriamente distribuídos entre essas G páginas, e que a probabilidade de dois campos apontadores (de base ou das estruturas) distintos designarem nós na mesma página é desprezível.

Suporemos também que a cada instante o sistema pode

manter simultaneamente na memória rápida um número de páginas razoável, embora muito menor que G . Admitiremos que as transferências só ocorrem quando uma falha é causada pelo algoritmo, e que a escolha da próxima página a ser recolocada na memória lenta leva em conta, a medida do possível, o funcionamento do mesmo.

Por exemplo, podemos supor, no algoritmo A3.1, que o sistema mantém na memória rápida a página que contém o nó $E+$, enquanto a marca de cada um de seus filhos (que estarão geralmente em páginas distintas) é examinada. Portanto, com memória rápida suficiente para duas páginas, o número de falhas geradas é menor ou igual a $(P_B + P_A) + N_A$, sendo o primeiro termo causado pelo exame das marcas, e o segundo pelo exame dos campos de cada nó.

É possível diminuir sensivelmente esse número de falhas, com ligeiras modificações no algoritmo. Uma solução imediata é armazenar na fila, em vez dos endereços dos nós, os conteúdos dos mesmos; desta forma poupam-se N_A falhas de páginas.

Uma solução mais praticável, por não consumir tanto espaço na pilha, consiste em adiar o teste da marca para quando o endereço do nó for retirado de PE (algoritmo A3.2). O número de acessos reduz-se igualmente a $P_A + P_B$, às custas de um aumento no número de entradas na fila.

No algoritmo A3.2, a fila pode chegar a conter $P_B + P_A - \lceil P_A / \tilde{p} \rceil$ entradas, em vez de $N_A - \lceil N_A / \tilde{p} \rceil$. Se as estruturas forem árvores disjuntas, esses limites são iguais; caso contrário, o primeiro é sempre maior.

Note-se que estamos supondo, em ambos os casos, que a fila é mantida toda na memória rápida, de modo que seu uso não gera falhas adicionais. Mesmo quando ela compartilha o

procedimento MARCA-HORIZONTAL-COM-MENOS-FALHAS:

efetua

	[área PE: fila de $P_B + N(\bar{p}-1)$ apontadores, inicialmente vazia;
1		repete, para J desde 1 até C_B ,
C_B		[se $R[J]$ é apontador, então
P_B		[PE $\leftarrow R[J]$;
		área E: apontador;
		repete, enquanto PE não é vazia,
$P_A + P_B$		[E \leftarrow PE;
		se E↓.marca é 'inútil', então
N_A		[E↓.marca \leftarrow 'útil';
		repete, para I desde 1 até E↓.c,
C_A		[se E↓.corpo[I] é apontador, então
P_A		[PE \leftarrow E↓.corpo[I];

Algoritmo A3.2 - Marcação Horizontal com menos falhas de páginas.

mecanismo de memória virtual, entretanto, as falhas de páginas causadas por ela são em geral desprezíveis, face ao total.

O uso de uma pilha, em lugar da fila PE, diminui um pouco o número de falhas, pois assim o nó E+ será frequentemente o que teve sua marca consultada por último. Uma redução maior pode ser conseguida se, na retirada de endereços de PE, for dada preferência aos que apontarem para páginas que estejam presentemente na memória rápida. A economia de transferências que isso proporciona, embora difícil de ser quantificada, pode ser considerável, pois constata-se na prática que a probabilidade de um nó ter um ou mais filhos na mesma página é bastante elevada.

3.7 Marcação com Pilha Distribuída

A pilha ou fila do algoritmo A3.1 pode ser substituída por um ítem auxiliar em cada nó, com tamanho suficiente para conter um endereço. Os nós já marcados que ainda não tiveram seus campos examinados são ligados entre si por essa área auxiliar. Se desprezarmos os demais apontadores desses nós, os mesmos formarão uma lista ligada, que pode ser usada como pilha ou como fila, sem maiores dificuldades {Knu 68 - 2.2.3}

Esta versão do algoritmo A3.1 não tem mais o problema de encontrar espaço suficiente para a pilha PE. Pelo contrário, seu problema é o desperdício de espaço, pois N áreas auxiliares devem ser reservadas, contra as $N_A - \lceil N_A / \bar{p} \rceil$ do algoritmo A3.1.

Este algoritmo é recomendável quando idiossincrasias da máquina ou da aplicação obrigarem a reservar em cada nó uma área de tamanho suficiente, que não esteja em uso no mo-

modos ligação: união uniforme disjunta de ('nada', apontador),

nô: [marca: marca-de-nô,
prox: ligação,
c: inteiro de 1 a \hat{c} ,
corpo: sequência de c campos];

procedimento MARCA-COM-PILHA-DISTRIBUIDA:

recebendo área L: ligação {que apontará para a lista de todos os
nôs marcados},

efetua

	[área PE: ligação, inicialmente 'nada';
	repete, para J desde 1 até C_B ,	
C_B	[se $R[J]$ é apontador
P_B		e $R[J].\downarrow.marca$ é 'inútil', então
N_{A1}	[$R[J].\downarrow.marca \leftarrow 'útil';$
	[$R[J].\downarrow.prox \leftarrow PE; PE \leftarrow R[J];$
]	áreas E, A: apontadores;
N_A	repete, enquanto $PE \neq 'nada'$,	
	[$E \leftarrow PE; PE \leftarrow E.\downarrow.prox;$
		$E.\downarrow.prox \leftarrow L; L \leftarrow E;$
	repete, para I desde 1 até $E.\downarrow.c$,	
C_A	[se $E.\downarrow.corpo[I]$ é apontador, então
P_A		$A \leftarrow E.\downarrow.corpo[I];$
		se $A.\downarrow.marca$ é 'inútil', então
N_{A+}	[$A.\downarrow.marca \leftarrow 'útil';$
	[$A.\downarrow.prox \leftarrow PE; PE \leftarrow c$
]	
]	

Algoritmo A3.3 - Marcação horizontal com pilha distribuída.

mento da recuperação. Algumas implementações da linguagem LISP 1.5, em computadores com palavras grandes, por exemplo, tem encontros adequados em cada nó.

Na versão apresentada a aqui (A3.3) a área PE é o endereço do primeiro nó marcado e não examinado, e prox a área auxiliar em cada nó. Como um bônus adicional, o algoritmo A3.3 encadeia pelo campo prox, ao fim da marcação, todos os nós da estrutura, e devolve o endereço do primeiro nó da cadeia no parâmetro L. Caso isso não seja conveniente no contexto em que a marcação é realizada, basta eliminar o argumento L e as instruções $L \leftarrow \text{'nada'}$ e $E \leftarrow \text{prox} \leftarrow L; L \leftarrow E$ de A3.3.

O número de falhas de páginas com este algoritmo é no máximo $(P_B + P_A) + N_A$, como no algoritmo A3.1. A otimização que resultou no algoritmo A3.2, infelizmente, não pode ser aplicada neste caso.

3.8 Marcação com Pilha Insuficiente

Quando não há espaço suficiente para uma pilha com $N_A - \lceil N_A / \bar{p} \rceil$ entradas, pode-se recorrer ao algoritmo A3.4. Este algoritmo usa uma pilha PE com um número reduzido k de entradas, e funciona como A3.1 enquanto houver espaço nela.

Quando a pilha PE estiver lotada, e for necessário inserir outro endereço, este é simplesmente ignorado. Portanto, quando a pilha PE ficar vazia, ainda poderá haver nós marcados com filhos não marcados. O algoritmo A3.4 procura localizar esses nós percorrendo sequencialmente o espaço de alocação; quando um candidato provável é encontrado, seu endereço é empilhado em PE, e seus descendentes não marcados (se os houver) serão percorridos, repetindo-se todo o processo.

procedimento MARCA-COM-PILHA-LIMITADA:

efetua

1	[áreas PE: pilha de k apontadores inicialmente vazia,	
		LESQ: apontador inicialmente LIM;	
		repete, para J desde 1 até C_B ,	
C_B		[se $R[J]$ é apontador,
P_B			e $R[J].\text{marca}$ é 'inútil', então
N_{A1}			$R[J].\text{marca} \leftarrow \text{'útil'}$;
			$PE \Leftarrow R[J]$;
		áreas E,A: apontadores;	
1]	repete,
K_3+1			enquanto PE não está vazia ou $LESQ < LIM$,
K_3	[se PE está vazia, então
K_3+K_1-N			$E \leftarrow LESQ$;
			repete
K_2			$LESQ \leftarrow LESQ + \text{tam}(LESQ)$
	até que $LESQ \geq LIM$ ou $LESQ.\text{marca} = \text{'útil'}$		
	senão		
$N-K_1$	$E \Leftarrow PE$;		
K_3	repete, para J desde 1 até $E.c$,		
$\tilde{c}.K_3$	se $E.c.\text{corpo}[J]$ é apontador, então		
$\tilde{p}.K_3$	$A \leftarrow E.c.\text{corpo}[J]$;		
	se $A.\text{marca}$ é 'inútil', então		
N_{A+}	[$A.\text{marca} \leftarrow \text{'útil'}$;
			se PE está cheia, então
K_1		$LESQ \leftarrow \min(LESQ, A)$	
$N_{A+}-K_1$		senão	
	$PE \Leftarrow A$		

Algoritmo A3.4 - Marcação horizontal com pilha limitada. Os parâmetros K_1 , K_2 e K_3 dependem da estrutura, e são comentados no texto.

A variável LESQ destina-se a tornar mais eficiente a localização dos nós descartados sem exame. No início de cada iteração da segunda malha de A3.4, garante-se que todo nó ativo está marcado, ou é marcável a partir de um elemento de PE, ou a partir de um campo de um nó marcado com endereço \geq LESQ.

É difícil dar uma análise precisa da eficiência de A3.4. Pode-se dizer apenas que é bastante inferior à dos vistos anteriormente. Se a pilha PE for muito reduzida, os parâmetros K_2 e K_3 podem chegar no pior caso a $O(N \cdot N_A)$ e $O((N_A)^2)$, respectivamente. Mesmo com uma pilha pouco menor que a necessária para A3.1, o algoritmo A3.4 pode ser obrigado a percorrer muitas vezes todos os nós do espaço.

Este algoritmo tende a produzir muito mais falhas do que A3.1, o que torna seu uso desaconselhável em sistemas de memória virtual. De qualquer forma, nestes sistemas o tamanho da pilha PE não é um problema.

3.9 Algoritmos Verticais

Na segunda classe de algoritmos de percurso e marcação, que chamaremos de algoritmos verticais, os filhos de um mesmo nó da floresta de percurso podem ocorrer, na ordem de marcação, intercalados com outros nós.

Após marcar um dos filhos de um dado nó, esses algoritmos podem suspender o exame dos demais campos deste, prosseguir examinando outros nós, e retomar mais tarde o exame do nó "suspenso". Para tanto, o algoritmo precisa "lembrar" o endereço do nó, e quais de seus campos já foram examinados.

O protótipo dos algoritmos verticais (A3.5) também usa uma pilha PE, cada entrada da qual contém o endereço de um nó, e uma pilha paralela PI cujas entradas são inteiros de 1 a \hat{c} . Cada um desses inteiros é o índice do último campo examinado do nó correspondente em PE. PE e PI funcionam na verdade como uma única pilha, PEI.

No algoritmo A3.5, a ordem de marcação dos nós de uma estrutura pode ser definida da seguinte maneira, a partir da árvore de percurso: a raiz da árvore, seguida da concatenação da ordem de percurso de suas sub-árvores. Esta é a "pré-ordem", definida por Knuth [Knu 68 - 2.3.2], exceto pelo fato das sub-árvores serem percorridas na sequência inversa (da última para a primeira).

No início de cada iteração de MALHA, o conteúdo das pilhas PE e PI, lido do fundo para o topo, descreve os nós e campos da cadeia que, na árvore de percurso, vai de R ao nó E+.

Conclui-se portanto que o número máximo de entradas simultaneamente ocupadas de PEI é um a menos da profundidade da árvore de percurso, e portanto sempre menor que N.

A utilização de uma fila, em lugar da pilha PEI, não traz nenhuma vantagem importante, e sua implementação é um pouco mais complexa. Portanto, não nos preocuparemos com essa possibilidade.

O número de falhas causadas pelo algoritmo A3.5 pode ser limitado por $P_A + P_B + N_{A+}$, sendo que, destas, N_{A+} ocorrem nos acessos a nós que acabaram de ser desempilhados de PE. Na verdade, muitos destes acessos não causam falhas de páginas, graças à disciplina de pilha usada em PE. Neste aspecto, portanto, o algoritmo A3.5 é comparável ao de marcação horizontal A3.1.

procedimento MARCA-VERTICALMENTE:

efetua

1 [áreas PE: pilha de N-1 apontadores, inicialmente vazia,
 PI: pilha de N-1 inteiros de 1 a \hat{c} , inicialmente vazia;
 repete, para J desde 1 até C_B ,
 se $R[J]$ é apontador,
 e $R[J].\text{marca}$ é 'inútil', então
 áreas E, A: apontadores,
 I: inteiro de 1 a \hat{c} ;

 $E \leftarrow R[J]$;
 MALHA: repete indefinidamente
 $E.\text{marca} \leftarrow \text{'útil'}$;
 $I \leftarrow 1$;
 PROCURA-PROXIMO: repete indefinidamente
 se $E.\text{corpo}[I]$ é apontador, então
 $A \leftarrow E.\text{corpo}[I]$;
 se $A.\text{marca}$ é 'inútil', então
 termina PROCURA-PROXIMO;
 repete,
 enquanto $I = E.c$,
 se PE estiver vazia, então
 termina MALHA;
 $I \leftarrow PI$; $E \leftarrow PE$;
 $I \leftarrow I+1$;
 EMPILHA:
 $PE \leftarrow E$; $PI \leftarrow I$;
 PROSSEGUE:
 $E \leftarrow A$

Algoritmo A3.5 - Marcação vertical com pilha.

3.10 A Variante de Clark

É possível melhorar o uso da pilha, no algoritmo A3.5, evitando inserir nela endereços de nós que não contêm mais campos apontadores a examinar. Esta variante (A3.6) parece ser devida a D.W.Clark, que a usou num algoritmo para cópia de estruturas {Cla 6/76}.

A floresta de percurso e a ordem de marcação de A3.6 são as mesmas de A3.5. Todos os comandos de A3.5 são executados em A3.6 exatamente o mesmo número de vezes; excetuam-se os comandos de inserção/remoção de PE e PI (executados K_2 vezes contra N_{A+}), o teste "PE está vazia" (efetuado $N_{A1} + K_2$ vezes em vez de N_A), e, eventualmente, instruções de desvio. Note que os parâmetros K_1 e K_3 não influem no tempo total do algoritmo A3.6.

O parâmetro K_2 é o número de apontadores primários que são seguidos por campos apontadores, dentro do mesmo nó. Esse número é portanto sempre menor ou igual a N_{A+} . Embora em algumas estruturas ocorra a igualdade (p. ex., na da figura F3.4), em geral K_2 é sensivelmente menor. Por exemplo, se a estrutura original é uma floresta, K_2 é o número de folhas menos o número de raízes.

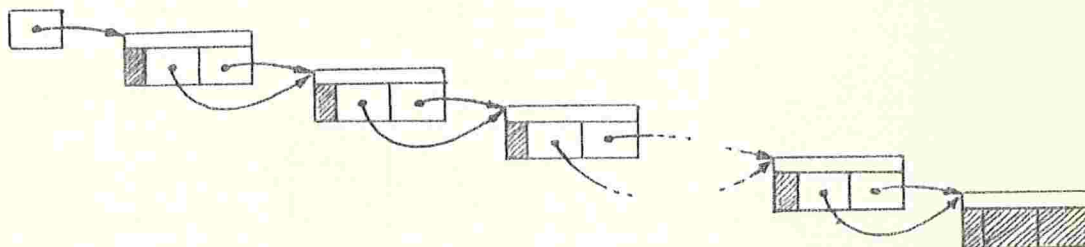


Figura F3.4 - Uma estrutura com $K_2 = N_{A+}$.

procedimento MARCA-VERTICAL-VARIANTE-DE-CLARK:
efetua

1	[áreas PE: pilha de N-1 endereços de nós inicialmente vazia;
		PI: pilha de N-1 inteiros de 2 a \tilde{c} , inicialmente vazia,
		repete, para J desde 1 até C_B ,
C_B		se $R[J]$ é apontador,
P_B		e $R[J].\downarrow.marca$ é 'inútil', então
N_{A1}		áreas A, E: apontadores,
		I: inteiro de 1 a \tilde{c} ;
		$E \leftarrow R[J]$;
		MALHA: repete indefinidamente
N_A		[$E.\downarrow.marca \leftarrow 'útil'$;
		$I \leftarrow 1$;
		PROCURA-PRÓXIMO:
		repete indefinidamente
$C_A - K_1$		[se $E.\downarrow.corpo[I]$ é apontador, então
$P_A - K_2$		[$A \leftarrow E.\downarrow.corpo[I]$;
$N_{A+} - K_3$	se $A.\downarrow.marca$ é 'inútil', então	
	termina PROCURA-PRÓXIMO;	
	repete,	
$C_A + K_2 - K_1 - N_{A+}$	enquanto $I = E.\downarrow.c$,	
$N_{A1} + K_2$	[se PE estiver vazia, então	
N_{A1}	termina MALHA;	
K_2	$I \leftarrow PI$; $E \leftarrow PE$;	
	$A \leftarrow E.\downarrow.corpo[I]$;	
	se $A.\downarrow.marca$ é 'inútil', então	
	termina PROCURA-PRÓXIMO;	
K_3	$I \leftarrow I + 1$;	
$C_A - K_1 - N_A$	EMPILHA-SE-NECESSÁRIO:	
N_{A+}	repete,	
$N_{A+} + K_1 - K_2$	enquanto $I < E.\downarrow.c$,	
K_1	[$I \leftarrow I + 1$;	
	se $E.\downarrow.corpo[I]$ é apontador, então	
K_2	[$PE \leftarrow E$; $PI \leftarrow I$;	
	termina EMPILHA-SE-NECESSÁRIO;	
N_{A+}	PROSSEGUE:	
	$E \leftarrow A$;	

Algoritmo A3.6 - A variante de Clark. K_1, K_2 e K_3 são parâmetros das estruturas.

Na variante de Clark, a pilha não descreve mais a cadeia que leva de R a $E+$. O número de entradas nas pilhas PE e PI é geralmente menor que com A3.5, e o tamanho das entradas de PI é ligeiramente menor (inteiro de 2 a \hat{c} em vez de inteiro de 1 a \hat{c}). Se $\hat{c} \leq 2$, a pilha PI pode ser dispensada, e o algoritmo simplifica-se bastante.

O número de falhas geradas por este algoritmo é no máximo $P_A + P_B + K_2$, menor que o de A3.5, pois cada nó é revisto só se contiver mais algum apontador a examinar.

3.11 O Algoritmo de Deutsch, Schorr e Waite

A grande desvantagem dos algoritmos verticais é a necessidade de uma pilha maior que a exigida pelos horizontais. Entretanto, em 1967 H. Schorr e W. M. Waite {Sch/Wai 8/67} publicaram uma versão de A3.5 que dispensa totalmente a pilha PE. Esse algoritmo (A3.7) foi descoberto independentemente por L. Peter Deutsch, na mesma época.

Suponha que $\epsilon_1, \epsilon_2, \dots, \epsilon_n$ são (do fundo para o topo) os endereços de nós armazenados na pilha PE, em algum ponto da execução de A3.5. Como dissemos, existe um campo de ϵ_i que aponta para ϵ_{i+1} , onde $i=1, 2, \dots, n-1$, e ϵ_n aponta para $E+$. A idéia do algoritmo de Deutsch, Schorr e Waite é substituir temporariamente o conteúdo desse campo do nó ϵ_i por ϵ_{i-1} , para $i=2, 3, \dots, n$.

Desta forma, conhecido ϵ_n e o índice do campo invertido, ϵ_{n-1} pode ser determinado; e quando ϵ_n tiver que ser retirado da pilha, seu campo invertido pode ser restaurado com o valor de E . Como nessa ocasião E passa a conter ϵ_n , as mesmas operações podem ser repetidas para retirar ϵ_{n-1} , e assim por diante. A pilha PE então é supérflua, sendo substi

procedimento DEUTSCH-SCHORR-WAITE:
efetua

1	[áreas PI: pilha de N-1 inteiros de 1 a \hat{c} , inicialmente vazia, PE: apontador,
		repete, para J desde 1 até C_B ,
C_B		se $R[J]$ é apontador,
P_B		e $R[J].\text{marca}$ é 'inútil', então
N_1		áreas A, E: apontadores, I: inteiro de 1 a \hat{c} ;
		$E \leftarrow R[J]$;
		MALHA: repete indefinidamente
N_A		$E.\text{marca} \leftarrow \text{'útil'}; I \leftarrow 1$;
		PROCURA-PROXIMO: repete indefinidamente
C_A		se $E.\text{corpo}[I]$ é apontador, então
P_A	[$A \leftarrow E.\text{corpo}[I]$;	
N_{A+}	se $A.\text{marca}$ é 'inútil', então	
$C_A - N_{A+}$	termina PROCURA-PROXIMO;	
C_A	repete,	
N_A	enquanto $I = E.c$,	
N_{A1}	[se PI é vazia, então	
	termina MALHA;	
	DESEMPILHA:	
N_{A+}	$I \leftarrow PI$;	
	$A \leftarrow E; E \leftarrow PE$;	
	$PE \leftarrow E.\text{corpo}[I]$;	
	$E.\text{corpo}[I] \leftarrow A$;	
	$I \leftarrow I + 1$;	
$C_A - N_A$	EMPILHA:	
N_{A+}	$E.\text{corpo}[I] \leftarrow PE; PE \leftarrow E$;	
	$PI \leftarrow I$;	
	PROSSEGUE:	
	$E \leftarrow A$;	

Algoritmo A3.7 - O marcador de Deutsch-Schorr-Waite, com a pilha PI explícita.

pela "inversa" da cadeia que levaria de R até $E \downarrow$ (fig. F3.5).

Quando o algoritmo termina, todos os campos invertidos nesse processo terão sido restaurados, e a estrutura estará intacta (porém marcada).

Note-se que o tamanho total da pilha PI não precisa ser maior que $(N-1)\log_2 \hat{c}$. Para o caso do LISP 1.5, por exemplo, que tem $\hat{c} = 2$, cada entrada fica reduzida a um bit. Note-se ainda que, para um valor fixo de M , o tamanho total de PI diminui quando \hat{c} aumenta.

Se o número de campos variar bastante de um nó para outro, pode ser conveniente usar entradas de tamanho variável. A entrada correspondente ao nó E é um inteiro de 1 a $E \downarrow.c$, e pode ser codificada usando apenas $\log_2 E \downarrow.c$ bits.

Uma economia adicional pode ser obtida notando-se que os índices em PI correspondem sempre a campos não atômicos do nó. Em vez de I , pode-se empilhar o número I' de apontadores entre os I primeiros campos. O intervalo de variação de I' é bem menor que o de I , ocupando portanto menos espaço em PI .

Note-se que se um nó não tem filhos, ele nunca é empilhado; e se tiver um único filho, sua entrada I' em PI só pode ser 1 , e portanto ocupa zero bits.

Naturalmente, esse esquema só é praticável se os nós tiverem em geral poucos campos, ou se todos os campos apontadores ocorrerem em posições consecutivas; caso contrário, o tempo gasto para localizar o I' -ésimo apontador dentro do nó pode ser considerável.

Essas otimizações, e o uso de uma pilha PI separada, são devidas a B. Wegbreit {Weg 9/72}. A versão original

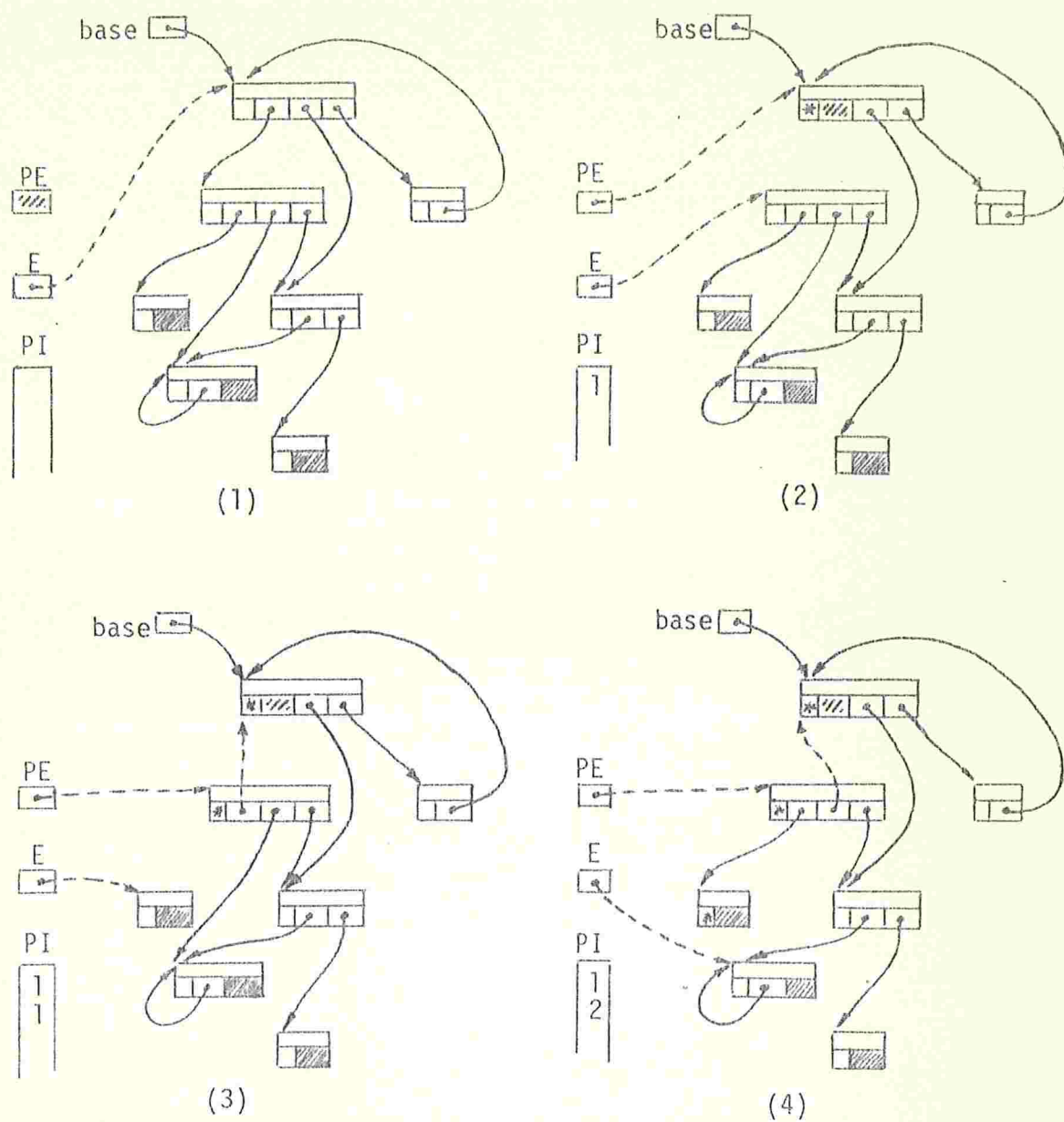


Figura F3.5 - Exemplo do funcionamento do algoritmo de Deutsch, Schorr e Waite (A3.7). De (1) a (7) - situação no início de cada iteração de MALHA; (8) - situação final. (continua)

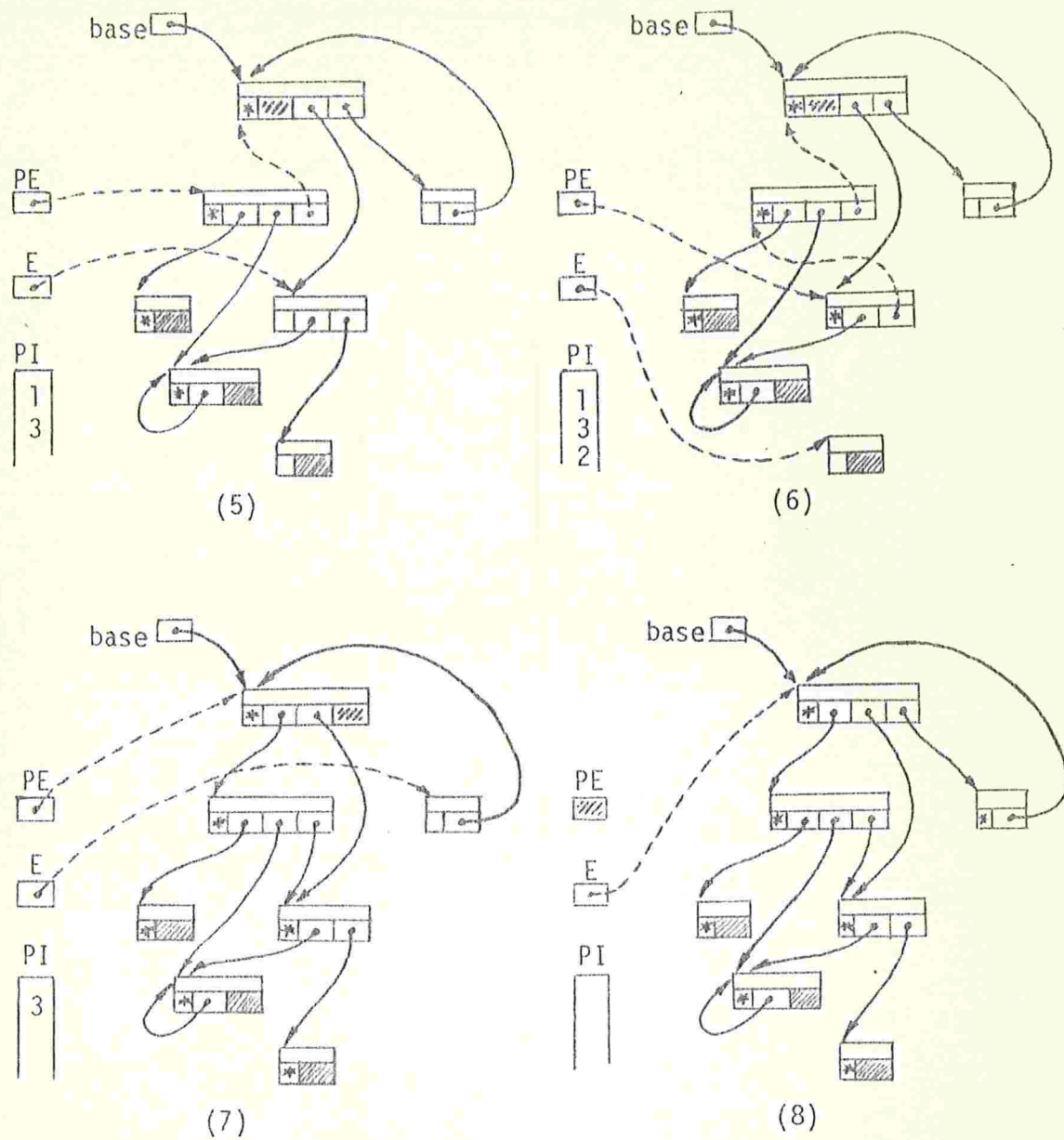


Figura F3.5 (conclusão)

de Deutsch, Schorr e Waite substituía a pilha PI por um campo adicional ind:inteiro de 1 a c em cada nó, para guardar o índice do campo invertido (se fosse o caso) desse nó. Isto é, a pilha PI não existia, e os comandos $I \leftarrow PI$ e $PI \leftarrow I$ eram substituídos por $I \leftarrow PE \downarrow .ind$ e $PE \downarrow .ind \leftarrow I$, respectivamente. Essa versão é aconselhável quando houver espaço disponível em cada nó, devido a idiossincrasias da máquina e da aplicação. Note-se que se $\hat{c} = 2$, como era o caso da versão original de A3.7, o campo ind ocupa um único bit.

O número de falhas de páginas do algoritmo de Deutsch-Schorr-Waite é o mesmo do algoritmo vertical com pilha A3.5, isto é, menos de $(P_B + P_A) + N_{A+}$.

3.12 Marcação de Nós Amorfos e Sobrepostos

Em todos os algoritmos de marcação anteriores, supusemos nós disjuntos dois a dois e auto-descritos. Se os nós forem amorfos, todo apontador deverá conter, além do endereço do nó apontado, um descritor que dê o formato do mesmo. Neste caso, os algoritmos vistos até agora podem ser usados, com ligeiras modificações: basta carregar sempre, junto com cada endereço de nó, o descritor correspondente, e consultar este sempre que necessário.

Nós amorfos, porém, são geralmente usados em implementações que permitem nós sobrepostos; e esta é uma característica mais problemática. A própria idéia de marcar os nós atingíveis tem que ser abandonada, pois não é possível incluir a marca como um item do nó.

Em vez de uma marca em cada nó, pode-se associar uma a cada campo. O algoritmo de percurso deve marcar todos

os campos dos nós atingíveis; campos que não estiverem marcados, ao fim do mesmo, poderão ser recolhidos, ao espaço livre. Esta coleta, exige um exame sequencial de todos os campos, e, por conseguinte, exige que os mesmos sejam auto-descritos.

Muitos dos algoritmos que vimos nas seções anteriores podem ser adaptados para esta situação. A título de exemplo, daremos apenas as versões dos algoritmos horizontal (A3.2) e o de Deutsch-Schorr-Waite (A3.7). Estas versões são dadas a seguir (A3.8 e A3.9).

Comparado com a versão original (A3.2), uma característica do algoritmo A3.8 é que, ao encontrar um novo apontador acessível, ele é obrigado a examinar todos os campos do nó apontado, quer esse nó já tenha sido visitado quer não. Portanto, a eficiência fica um pouco comprometida; a indexação de um campo é executada C_{A*} vezes (em lugar de C_A), o teste da marca é efetuado C_{A*} vezes (contra N_A), e sua atribuição C_A vezes (em lugar de N_A). O parâmetro C_{A*} é o número aparente de campos acessíveis, isto é, a soma de E.c para todo apontador acessível ou de base E. Esse total é aproximadamente $(P_A + P_B) \cdot \tilde{c}$.

Observações análogas valem para todos os outros algoritmo que podem ser adaptados a estruturas com nós superpostos. Nos algoritmos verticais, o tamanho da pilha tende a aumentar, pois nela pode ser inserido várias vezes o endereço de um mesmo nó. Assim, por exemplo, no algoritmo de Deutsch-Schorr-Waite modificado (A3.9) o tamanho da pilha PI só pode ser limitado, na prática, por P. A estrutura da fig. F3.6 é um exemplo que exige P_A entradas.

No algoritmo A3.9, a pilha PI não pode mais ser substituída por um item extra em cada nó, pelas mesmas ra-

modos campo: [marca: marca-de-não,
 núcleo: união disjunta de (átomo, apontador)];
 apontador: [c: inteiro de 1 a \hat{c} ,
 en: endereço de sequência de c campos];

procedimento MARCA-HORIZONTALMENTE-NÓS-AMORFOS:

1	[área PE: pilha de $P_B + N(1 - \hat{p})$ apontadores, inicialmente vazia;
C_B		repete, para J desde 1 até C_B ,
P_B		[se $R[J].\text{núcleo}$ é apontador, então
		$PE \leftarrow R[J]$;
$P_A + P_B$		repete, enquanto PE não é vazia,
C_{A*}		$E \leftarrow PE$;
C_A		repete, para I desde 1 até E.c,
	se $E.\text{ent}[I].\text{marca}$ é 'inútil', então	
P_A	[$E.\text{ent}[I].\text{marca} \leftarrow \text{'útil'}$;	
	se $E.\text{ent}[I].\text{núcleo}$ é apontador, então	
	$PE \leftarrow E.\text{ent}[I].\text{núcleo}$;	

Algoritmo A3.8 - Marcação horizontal de nós sobrepostos.

O parâmetro C_{A*} é definido no texto.

modos campo: [marca: marca-de-não,
 núcleo: união disjunta de (átomo, apontador)],
 apontador: [c: inteiro de 1 a \hat{c} ,
 en: endereço de sequência de c campos];

procedimento D-S-W-PARA-NÓS-AMORFOS:
 efetua

1	áreas PI: pilha de C inteiros de 1 a \hat{c} , inicialmente vazia, PE: apontador, I: inteiro de 1 a \hat{c} , A, E: apontadores;
	repete, para J desde 1 até C_B , se $R[J]$.núcleo é apontador, então
C_B	
P_B	$E \leftarrow R[J].núcleo$;
	MALHA:
	repete indefinidamente
$P_A + P_B$	$I \leftarrow 1$;
	PROCURA-PRÓXIMO:
	repete indefinidamente
C_{A*}	se $E.en\downarrow[I].marca$ é 'inútil', então
C_A	$E.en\downarrow[I].marca \leftarrow 'útil'$;
	se $E.en\downarrow[I].núcleo$ é apontador, então
P_A	$A \leftarrow E.en\downarrow[I].núcleo$
	termina PROCURA-PRÓXIMO;
$C_{A*} - P_A$	repete,
C_{A*}	enquanto $I = E.c$,
$P_A + P_B$	se PI é vazia, então
P_B	termina MALHA;
	DESEMPILHA:
P_A	$I \Leftarrow PI$;
	$A \leftarrow E$; $E \leftarrow PE$;
	$PE \leftarrow PE.en\downarrow[I].núcleo$;
	$PE.en\downarrow[I].núcleo \leftarrow A$;
$C_{A*} - P_A - P_B$	$I \leftarrow I + 1$;
P_A	EMPILHA:
	$E.en\downarrow[I].núcleo \leftarrow PE$;
	$PE \leftarrow E$; $PI \Leftarrow I$;
	PROSSEGUE:
	$E \leftarrow A$

Algoritmo A3.9 - Marcador de Deutsch, Schorr e Waite para nós amorfos e superpostos. C_{A*} é definido no texto.

ções que não se pode associar a este uma marca. Entretanto, é possível substituí-la por um item extra em cada apontador, ao lado do item c , sem nenhuma dificuldade.

O número de falhas do algoritmo A3.8 é o mesmo da versão original, $P_A + P_B$. No algoritmo A3.9, o número de falhas é menos de $(P_B + P_A) + P_A$, se a memória rápida for suficiente para duas ou mais páginas.

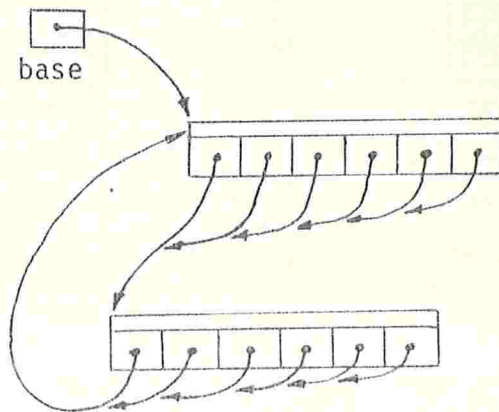


Figura F3.6 - Uma estrutura que exige P_A entradas na pilha de A3.9.

3.13 Marcação por Segmentos

Campos amorfos não trazem por si próprios nenhuma dificuldade adicional ao problema de marcação: as técnicas aplicáveis a nós amorfos podem ser facilmente estendidas a esse caso.

Entretanto, implementações com campos amorfos são geralmente recomendáveis quando os campos podem ter tamanhos muito reduzidos (booleanos, inteiros pequenos, etc.) e o nú-

mero de campos é muito maior que o de apontadores. Nestas condições, incluir um descritor em cada campo é um desperdício considerável de espaço, e o mesmo pode ser dito do uso de uma marca individual em cada campo.

Um problema mais sério é a impossibilidade de percurso sequencial dos campos. Como veremos, tais percursos são indispensáveis na etapa de coleta, e portanto a marcação só terá utilidade se esse problema puder ser contornado de alguma forma.

Uma solução, sugerida por B. Wegbreit {Weg 3/72}, consiste em ignorar parcialmente as fronteiras dos campos, e dividir o espaço de alocação em segmentos de tamanho uniforme, sendo a marcação feita ao nível destes. Isto é, cada segmento recebe uma única marca, que é ligada se e somente se o mesmo contiver algum nó ou campo ativo, ou partes deles.

Desta forma, o número de marcas é mantido dentro de limites razoáveis, e, na coleta, o percurso sequencial da memória pode ser feito segmento-a-segmento. O preço pago por esta solução é um certo desperdício de espaço, pois uma área inativa situada no mesmo segmento que uma ativa não será recolhida.

Além de serem usadas na coleta, as marcas tem uma segunda finalidade, como já ressaltamos: indicar ao algoritmo marcador quais os nós ou apontadores já examinados, para evitar circularidades e repetições inúteis.

Se houver dois ou mais apontadores num mesmo segmento, entretanto, a marca deste último não permitirá determinar qual dos dois já foi percorrido. Portanto, para que esse teste possa ser feito, é preciso usar uma outra marca em cada apontador, ou garantir (pela escolha adequada do tamanho dos segmentos, ou pelo uso de enchimentos convenientes) que cada

apontador ocupe pelo menos um segmento completo.

Com estas precauções, não é difícil adaptar os algoritmos já vistos à marcação por segmentos. Por exemplo, para adaptar o algoritmo de Deutsch-Schorr-Waite (A3.7) para marcação por segmentos, precisaremos redefinir os modos campo e apontador como segue:

```
modos campo: união de (átomo, apontador),
    apontador: [desc: descritor,
                en: endereço de nó]
```

onde um dado de modo descritor consiste de uma descrição do número de campos de um nó, e dos modos destes, oportunamente codificada.

Toda expressão da forma $\epsilon.en \uparrow [I].n\acute{u}cleo$ deve ser substituída pela operação $seleciona(\epsilon.en, \epsilon.desc, I)$ que devolve o I -ésimo campo (com modo correto) do nó de endereço $\epsilon.en$ e descritor $\epsilon.desc$. A expressão $\epsilon.c$ deve ser substituída por $num-campos(\epsilon.desc)$, que dá o número de campos de um nó com descritor $\epsilon.desc$.

Se o tamanho de cada segmento for ℓ , as marcas podem ser representadas por um vetor marcas: sequência de M/ℓ marcas-de-nós, sendo $marca[(\epsilon - INI)/\ell]$ a marca do segmento de endereço ϵ .

O teste da marca de um campo deve ser substituído pela conjunção das marcas de todos os segmentos sobrepostos ao campo. A marcação do campo consistirá então em tornar 'úteis' todas essas marcas.

O tamanho do segmento deve ser o menor possível, para diminuir ao máximo o desperdício de espaço causado por seg

mentos parcialmente inativos. O espaço disponível para o vetor de marcas, o tempo gasto em testá-las e modificá-las, e as peculiaridades dos algoritmos de coleta, são alguns fatores que se opõem a essa consideração.

3.14 O Algoritmo Marcador de Zave

Uma solução diferente para o problema da marcação de estruturas com nós sobrepostos e campos amorfos é a oferecida pelo algoritmo de D.A. Zave {Zav 7/75}. Seu algoritmo exige uma área adicional em cada campo apontador, com tamanho suficiente para um endereço, e uma marca binária; em compensação, não exige nenhuma marca nos nós, nem nos campos atômicos. É portanto recomendável em implementações onde a relação P/C é pequena.

Em vez de marcar todos os nós ou campos ativos, o algoritmo de Zave marca apenas os apontadores acessíveis, e forma uma lista ligada com os mesmos. O algoritmo de coleta, também proposto por Zave, pode percorrer esta lista, em vez de efetuar o percurso sequencial da memória, como veremos no cap. IV.

Na descrição que apresentamos deste marcador (A3.10), suporemos campos auto-descritos, para maior simplicidade. A versão para campos amorfos pode ser facilmente obtida a partir de A3.10.

O algoritmo tem a mesma estrutura que o marcador horizontal A3.8. A pilha PE daquele é substituída por uma lista ligada linear, formada pelos apontadores para nós ainda não examinados, encadeados pelo item prox; PE contém o endereço do primeiro desses campos.

modos ligação: união uniforme disjunta de ('nada', endereço de apontador),

campo: união disjunta de (átomo, apontador),

apontador: $[c: \text{inteiro de } 1 \text{ a } \hat{c},$
 en: endereço de sequência de c campos,
 marca: marca-de-não,
 prox: ligação];

procedimento MARCADOR-DE-ZAVE:

recebendo área LAP: ligação,

efetua

1	área PE: ligação, inicialmente 'nada'; repete, para J desde 1 até C_B ,
C_B	se $R[J]$ é apontador, então
P_B	$[R[J].prox \leftarrow PE; PE \leftarrow end(R[J]);$ $LAP \leftarrow 'nada';$ áreas E,A: endereços de apontadores; repete
$P_B + P_A$	$E \leftarrow PE; PE \leftarrow E \downarrow .prox;$ $E \downarrow .prox \leftarrow LAP; LAP \leftarrow E;$ repete, para J desde 1 até $E \downarrow .c$,
C_{A*}	$A \leftarrow end(E \downarrow .en \downarrow [J]);$ se $A \downarrow$ é apontador
P_{A*}	e $A \downarrow .marca$ é 'inútil', então
P_A	$[A \downarrow .marca \leftarrow 'útil';$ $A \downarrow .prox \leftarrow PE; PE \leftarrow A$
	até que $PE = 'nada'$

Algoritmo A3.10- O marcador de Zave.

Os apontadores retirados da "pilha" são concatenados pelo mesmo item prox , à lista ligada dos apontadores acessíveis, o primeiro dos quais será $\text{LAP}+$. Note-se que esta lista contém os apontadores de base.

O parâmetro P_{A*} é definido, de maneira análoga à C_{A*} , como sendo a soma do número de apontadores em $E+$, para E percorrendo todos os apontadores de base e ativos; isto é, $P_{A*} \approx (P_A + P_B) \cdot \tilde{p}$.

O número de falhas deste algoritmo pode ser estimado em menos de $(P_B + P_A) + P_A$.

3.15 Outros Algoritmos de Marcação

Além dos algoritmos de marcação vistos nas seções anteriores, existem muitos outros destinados a estruturas com características peculiares, como florestas e estruturas sem circuitos. As particularidades das estruturas a marcar geralmente permitem aumentar a eficiência (de tempo e espaço) dos algoritmos.

Por exemplo, o algoritmo A3.11 a seguir, devido a G. Lindstrom [Lin 1/73] é aplicável quando as estruturas a marcar são árvores, e não exige nenhuma memória adicional; nem mesmo a marca em cada nó é consultada pelo algoritmo.

A marcação de cada nó $\text{ESTE}+$ é executada $\text{ESTE}+.c+1$ vezes; na primeira, ANT aponta para seu pai (ou é 'nada' se $\text{ESTE}+$ for raiz); em cada uma das seguintes, ANT aponta para uma sub-árvore de $\text{ESTE}+$ que acabou de ser marcada.

Após cada uma dessas marcações de $\text{ESTE}+$ o algoritmo retira o próximo campo a examinar, coloca em seu lugar o campo ANT , e permuta ciclicamente os campos de $\text{ESTE}+$. Na

```
modo campo: união uniforme disjunta de (átomo,
    apontador, 'nada');
```

procedimento MARCA-FLORESTAS-POR-ROTAÇÃO:

efetua

```

1
áreas ANT, PROX, ESTE: campos, inicialmente
                                'nada'
I: inteiro de 1 a  $\hat{c}$ ,
repete , para J desde 1 até  $C_B$ ,
se R[J] é apontador, então
    ESTE  $\leftarrow$  R[J];
    repete
        se ESTE é apontador, então
            ESTE.marca  $\leftarrow$  'útil';
            PROX  $\leftarrow$  ESTE $\downarrow$ .corpo[1];
            I  $\leftarrow$  1
            repete
                enquanto I < ESTE.c,
                    [ESTE $\downarrow$ .corpo[I]  $\leftarrow$  ESTE $\downarrow$ .corpo[I+1];
                    I  $\leftarrow$  I+1;
                [ESTE $\downarrow$ .corpo[I]  $\leftarrow$  ANT
            senão
                [PROX  $\leftarrow$  ANT;
                ANT  $\leftarrow$  ESTE;
                [ESTE  $\leftarrow$  PROX
        até que ESTE = 'nada'

```

Algoritmo A3.11 - Marcação de árvores por rotação.

primeira vez, essa operação equivale (a menos da permutação dos campos) à inversão do primeiro campo para incluir $ESTE\downarrow$ na cadeia invertida que leva da raiz a ele, tal como no algoritmo de Deutsch, Schorr e Waite.

A permutação dos campos garante que o próximo campo a examinar será sempre o primeiro do nó. Na última vez que o nó é visitado, o primeiro campo conterá o endereço do pai de $ESTE\downarrow$; a mesma operação descrita acima tem o efeito de restaurar completamente os campos de $ESTE\downarrow$, e o algoritmo retorna para o exame de seu pai (Figura F3.7). O algoritmo, portanto, não precisa saber quantos campos do nó já foram examinados, e portanto a pilha PI de A3.7 não é necessária.

A constante K_1 que ocorre na análise do algoritmo A3.11 é a soma de $(ESTE\downarrow.c) \cdot (ESTE\downarrow.c + 1)$, para todo nó atingível $ESTE\downarrow$. Se $ESTE\downarrow.c$ for constante ($=\hat{c}$) teremos $K_1 = (C_A + N_A)\hat{c}$. Como se pode ver, este método é conveniente apenas para estruturas com \hat{c} pequeno.

Este algoritmo é ótimo sob o ponto de vista de economia de espaço, mas, além de ser em geral mais demorado que os outros que vimos, está restrito a estruturas de árvores. Se houver cadeias paralelas (mas não fechadas), o algoritmo ainda é aplicável; porém, o número de vezes que um nó é marcado passa a ser multiplicado pelo número de cadeias que levam de R a ele, que pode crescer exponencialmente com N_A .

Se a estrutura tiver cadeias fechadas, A3.11 em geral termina antes de marcar todos os nós, e deixa de restaurar os nós alterados. Não se conhece nenhuma modificação que permita aplicá-lo a estruturas genéricas; mas tampouco se conhece uma demonstração de que isso seja impossível.

A marca não pode ser usada para evitar o exame de um mesmo nó pela segunda vez, pois o algoritmo é incapaz de

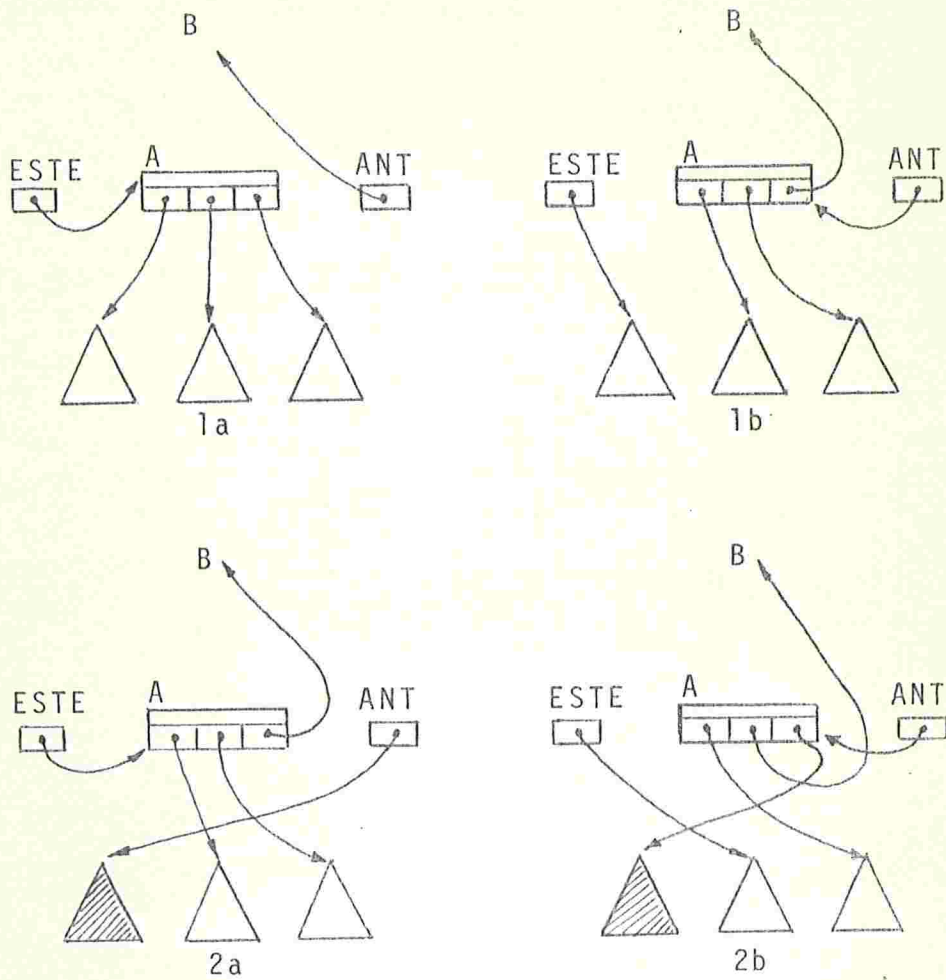


Figura F3.7 - Situações imediatamente antes da atribuição $ESTE \leftarrow \text{marca} \leftarrow \text{'útil'}$ (a), e após $ESTE \leftarrow \text{PROX}$ (b), no algoritmo A3.11, nas quatro "visitas" ao nó A, filho de B.
 Legenda: \triangle = sub-árvore a percorrer, \triangle = sub-árvore percorrida. (continua)

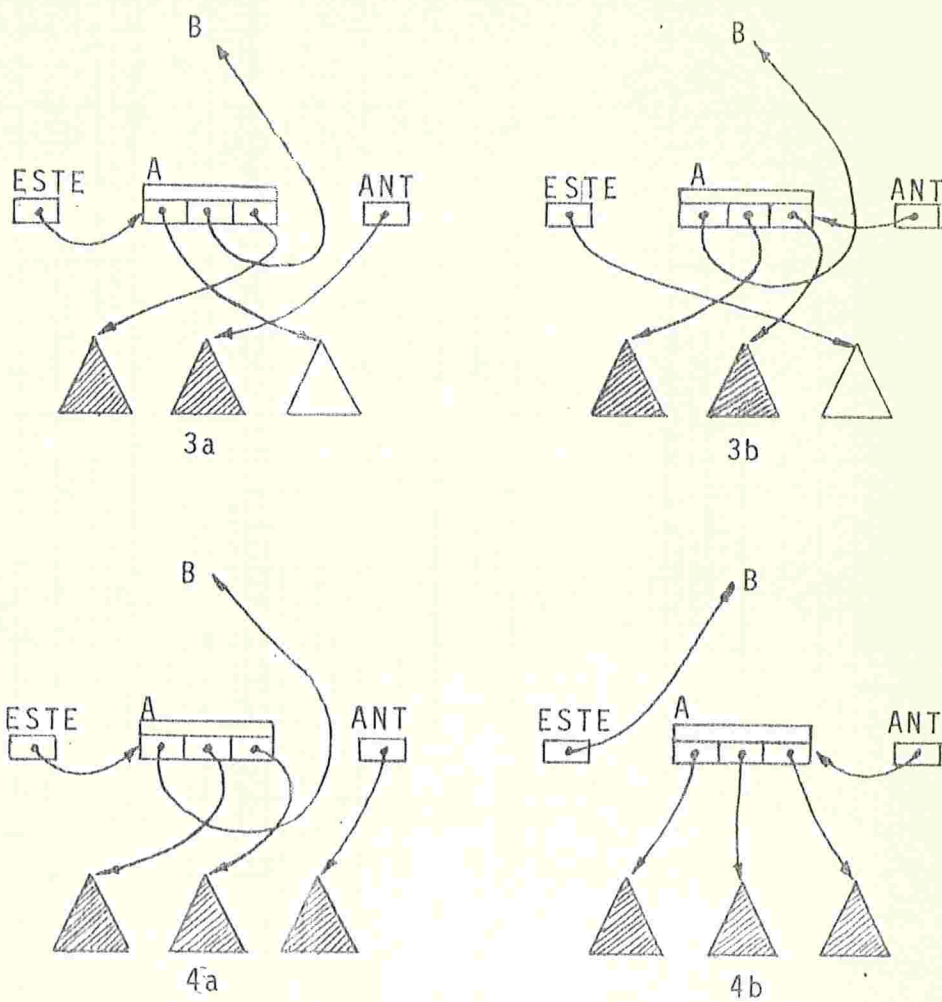


Figura F3.7 (conclusão)

decidir se o nó $ESTE\downarrow$ foi encontrado "descendo" na árvore (isto é, se ele é filho de $ANT\downarrow$) ou "subindo" (isto é, se $ANT\downarrow$ é filho dele). Assim, se $ESTE\downarrow.marca$ já for 'útil', é impossível dizer se se trata ou não de um caso de cadeias paralelas.

Essa distinção é possível, entretanto, se a estrutura for coerente, isto é, se os endereços dos filhos de um nó forem consistentemente maiores que o do próprio nó. Nestas condições, pode-se decidir se $ANT\downarrow$ era filho ou pai de $ESTE\downarrow$; se se tratar deste último caso, e $ESTE\downarrow.marca$ já for 'útil', trata-se de um nó já atingido por outra cadeia, e $ESTE$ pode ser tratado como um átomo por A3.11.

3.16 Algoritmos de Marcação: Conclusões

O algoritmo de Deutsch, Schorr e Waite (DSW) é provavelmente o mais popular dos algoritmos de percurso. Seu rival mais próximo parece ser o algoritmo de marcação horizontal com pilha (A3.1). Embora usando mais espaço, este algoritmo tem a vantagem de ser mais simples e um pouco mais rápido que o DSW.

Para uma comparação mais concreta dos dois algoritmos, tentamos codificar ambos na linguagem de máquina de um computador hipotético, com estrutura e repertório de instruções similares aos dos computadores reais mais comuns. Em ambos os casos foram usadas a mesma representação interna para os nós, e as mesmas técnicas de programação.

O tempo (em ciclos da máquina hipotética) necessário para a execução de cada um desses programas foi determinado como sendo

$$T_{HOR} = 3C_B + 3P_B + 6C_A + 3P_A + 13N_{A1} + 13N_{A+}$$

e

$$T_{DSW} = 3C_B + 3P_B + 6C_A + 3P_A + 6N_{A1} + 25N_{A+}$$

Se C_B, P_B e N_{A1} forem desprezíveis face a C_A, P_A e N_A , teremos

$$T_{HOR} \approx (13+6\tilde{c}+3\tilde{p}) N_A$$

$$T_{DSW} \approx (25+6\tilde{c}+3\tilde{p}) N_A$$

Para a linguagem LISP 1.5, por exemplo, onde $\tilde{c} = 2$ e \tilde{p} é tipicamente 1.8 {Cla/Gre 2/77}, temos $T_{DSW}/T_{HOR} \approx 1.4$.

O tempo T_{DSW} foi calculado para a variante original, com a pilha PI substituída por um campo extra em cada nó. Com a pilha PI separada, o tempo T_{DSW} seria ligeiramente maior: uns dois ou três ciclos adicionais por nó.

Em sistemas com memória virtual, onde em geral o espaço ocupado pela pilha não é um problema, o algoritmo horizontal A3.2 (ou similar) é o mais recomendável, por minimizar a movimentação de páginas.

Os algoritmos de marcação conhecidos exigem todos tempo no mínimo $O(N_A + P_A + P_B)$. Os que exigem no máximo tempo $O(N_A + C_A + C_B)$, como o DSW e o horizontal com pilha, exigem além disso, espaço auxiliar de tamanho $O(N_A)$.

Embora a necessidade de tempo $O(N_A + P_A + P_B)$ seja facilmente demonstrável, o mesmo não ocorre em relação ao espaço. Parece impossível efetuar a marcação de uma estrutura arbitrária, nesse tempo e usando um espaço auxiliar de tamanho apenas $O(1)$ ou $O(\log N_A)$, por exemplo. Entretanto, essa im-

possibilidade ainda não foi provada, e pode bem ser que essa impressão seja falsa.

CAPÍTULO IV - MÉTODOS DE COLETA E COMPACTAÇÃO

4.1 Coleta de Nós de Tamanho Uniforme

Recordemos que os recuperadores por marcação e coleta, após marcarem todos os nós ativos, percorrem o espaço de alocação e reincorporam os nós não marcados ao espaço livre.

Os algoritmos usados nessa etapa de coleta são bastante influenciados pela organização do espaço livre e pelo formato dos nós. Esses parâmetros também são determinantes para escolha do algoritmo alocador.

Quando os nós tem tamanho uniforme, o esquema mais simples consiste em dividir o espaço livre em áreas com o tamanho de um nó, e encadear essas áreas na forma de uma lista linear, a lista livre. A cada invocação, o algoritmo alocador pode obter a área de que necessita simplesmente retirando o primeiro elemento dessa lista.

A coleta dos nós inativos é igualmente simples: os nós do espaço de alocação são examinados sequencialmente, e cada nó não marcado é inserido nessa lista.

Neste método (algoritmo A4.1) suporemos que os nós são disjuntos, que o espaço de alocação é uma sequência de N nós, e que o procedimento MARCA torna 'úteis' as marcas de todos os nós ativos (excluindo os da lista livre), usando um algoritmo adequado dentre os vistos no capítulo III.

Normalmente, a marcação e recuperação são ativadas quando o alocador encontra a lista livre vazia. Não está excluída porém a possibilidade das mesmas serem iniciadas em

modos ligação: união uniforme disjunta de ('nada', apontador)

nó: [marca: marca-de-nó,

corpo: sequência de \hat{c} campos],

área LIVRE: ligação {primeiro nó da lista.livre}

procedimento COLETA-NÓS-UNIFORMES:

efetua

1	[área E: apontador;
		$E \leftarrow \text{INI};$
		repete,
N+1		enquanto $E < \text{LIM},$
N		se $E \downarrow \text{marca}$ é 'inútil', então
N-N _A		[move LIVRE para $E \downarrow$
		LIVRE $\leftarrow E;$
		senão
N _A		$E \downarrow \text{marca} \leftarrow \text{'inútil'};$
		$E \leftarrow E + \text{tam}(E \downarrow)$

procedimento ALOCA-NÓS-UNIFORMES:

recebendo área E: apontador,

efetua

[se LIVRE = 'nada', então
	MARCA-NÓS-ATIVOS;
	se LIVRE = 'nada', então erro{espaço esgotado};
	$E \leftarrow \text{LIVRE};$
	LIVRE $\leftarrow E \downarrow$ como ligação; $E \downarrow \text{marca} \leftarrow \text{'inútil'}$

Algoritmo A4.1 - Coleta e alocação de nós de tamanho uniforme.

outras circunstâncias (por exemplo, quando o programador ou as rotinas de execução desconfiarem que houve uma redução substancial no número de nós ativos).

O primeiro nó da lista livre é apontado pela área LIVRE, e cada um dos seguintes é apontado por um endereço armazenado no nó anterior.

A gerência da memória por lista livre tem a vantagem de permitir a recuperação imediata de um nó, quando for possível determinar com certeza que o mesmo se tornou inativo. Isso pode ocorrer com nós criados internamente pelas rotinas do sistema, fora do alcance do programador: tais nós tem frequentemente vida útil de duração conhecida, e portanto podem ser recolhidos à lista livre sem esperar pela próxima ativação do recuperador.

O número de falhas de páginas causadas por este algoritmo é igual ao número de páginas do espaço de alocação, G em vez do número de consultas a nós N_A . Isto é devido ao fato de que, no percurso sequencial da memória, cada nó consultado está na mesma página do nó anterior, exceto quando ele é o primeiro nó da página.

4.2 Coleta de Nós de Tamanhos Variados

Nós de tamanhos variados dificultam bastante a alocação e a coleta com lista livre. As áreas que fazem parte desta última terão também tamanhos variados; para criar um novo nó de tamanho t dado, é preciso localizar um elemento da lista livre com tamanho $t' \geq t$, "recortar" dele uma área de tamanho t , e deixar o restante (se houver) na lista livre.

Na coleta, deve-se determinar se cada nó não marcado é adjacente a outro nas mesmas condições, ou a uma área da

modos elemento: união disjunta de (\bar{n} o, \bar{n} o-livre);
 \bar{n} o-livre: [desc: descritor-de- \bar{n} o-livre,
 lixo: dado de tamanho desc.compr];
 ligação: união disjunta uniforme de ('nada', endereço
 de \bar{n} o-livre),
 descritor-de- \bar{n} o-livre: [prox: ligação,
 compr: inteiro de 0 a M];
 área LIVRE: ligação {primeiro elemento da lista livre};
 procedimento COLETA- \bar{N} OS- \bar{N} ÃO-UNIFORMES-COM-LISTA-LIVRE:

efetua

1	[áreas E: endereço de elemento, A: ligação,
		T: inteiro de 1 a M;
		E \leftarrow INI; LIVRE \leftarrow 'nada';
		repete,
N+N _L +1		enquanto E < LIM,
N+N _L		se E \dagger é \bar{n} o,
N		e E \dagger .marca é 'útil', então
N _A		[E \dagger .marca \leftarrow 'inútil'; E \leftarrow E + tam(E \dagger)
		senão
N+N _L -N _A		T \leftarrow tam(E \dagger);
	se LIVRE \neq 'nada',	
N+N _L -N _A -N _L *	e LIVRE + LIVRE \dagger .desc.compr = E, então	
	[LIVRE \dagger .desc.compr \leftarrow LIVRE \dagger .desc.compr + T	
	senão	
N _L *	A \leftarrow LIVRE;	
	LIVRE \leftarrow E;	
	LIVRE \dagger .desc \leftarrow [prox: A,	
	compr: T - tam(descritor-de- \bar{n} o-livre)];	
	E \leftarrow E + T	

Algoritmo A4.2 - Recuperação de \bar{n} os com tamanhos variados, usando lista livre. Os parâmetros N_L e N_{L*} são o número de elementos na lista livre, antes e depois da recuperação.

lista livre; em caso afirmativo, as duas áreas devem ser con-sadas numa só.

Existe uma grande variedade de algoritmos para a implementação de tais recuperadores e alocadores. O algoritmo A4.2 é um exemplo dos mais simples, e, por razões de espaço, será o único que aqui veremos. Outros algoritmos, e sua discussão, podem ser encontrados em {Knu 68 - 2.5}.

Em A4.2, o espaço de alocação é considerado uma sequência de N nós (marcados ou não), intercalados com zero ou mais elementos da lista livre. Esses dois tipos de áreas são supostos distinguíveis. O modo nô, aqui e no restante do capítulo, pressupõe, a menos que seja dito de outra forma, a declaração.

modo nô: [marca: marca-de-nô,
c: inteiro de 1 a \hat{c} ,
corpo: sequência de c campos]

Os elementos da lista livre são nôs-livres, cada um dos quais contém informação sobre seu tamanho e o endereço do elemento seguinte da lista.

O algoritmo alocador (A4.3) supõe que o tamanho do nô a alocar é pelo menos o de um nô-livre. Isto é necessário, para que o recuperador possa recolher à lista livre o nô alocado, quando este se tornar inativo.

Uma análise detalhada da eficiência do algoritmo alocador, e do comportamento dos parâmetros N_L e N_{L*} , está além das pretensões deste trabalho. Pode-se notar, entretanto, que o tempo para a alocação de um único nô não é constante, e pode ser considerável.

A lista livre pode conter inicialmente um único ele-

procedimento ~~ALOCA-NÓS-NÃO-UNIFORMES-COM-LISTA-LIVRE~~:

recebendo T: inteiro de tam(descriptor-de-nó-livre) a M {tamanho
do nó a alocar}

e área E: campo {que apontará para o nó alocado};

efetua

```

[ áreas A, UA: ligações;
  procedimento LOCALIZA-ELEMENTO-LIVRE-ADEQUADO:
    efetua
      [ UA ← 'nada'; A ← LIVRE,
        repete,
          enquanto A ≠ 'nada',
            e (T < A↓.desc.compr ou T ≠ tam(A↓));
            [ UA ← A;
              A ← A↓.prox;
        LOCALIZA-ELEMENTO-LIVRE-ADEQUADO;
      se A é 'nada', então
        [ MARCA-NÓS-ATIVOS;
          COLETA-NÓS-NÃO-UNIFORMES-COM-LISTA-LIVRE;
          LOCALIZA-ELEMENTO-LIVRE-ADEQUADO;
          se A é 'nada', então erro{não há espaço para este nó};
        se T = tam(A↓), então
          [ se UA = 'nada', então
              LIVRE ← A↓.desc.prox
            senão
              UA↓.desc.prox ← A↓.desc.prox;
          E ← A
        senão
          [ A↓.compr ← A↓.compr - T;
            E ← lim(A↓)
          ]
      ]

```

Algoritmo A4.3 - Alocação de nós de tamanho variado a partir de uma lista livre.

mento; porém, após as primeiras recuperações, ela tende a se fragmentar num grande número de elementos pequenos. Essa fragmentação, além de diminuir a eficiência do alocador, pode causar o término prematuro do programa por falta de espaço; pode ocorrer que nenhum dos elementos da lista livre seja suficientemente amplo para a alocação de um novo nó, embora a união de todos eles o seja.

Embora existam muitas técnicas para aumentar a eficiência do alocador, e diminuir a fragmentação da lista livre, todas as que se conhece produzem apenas melhoramentos marginais.

4.3 Algoritmos de Compactação

Em vista dos problemas que a gerência da lista livre apresenta para nós de tamanhos variados, usam-se em geral neste caso algoritmos de compactação para recuperar o espaço livre.

O objetivo desses algoritmos é deslocar todos os nós atingíveis de modo que eles ocupem áreas consecutivas num dos extremos da memória, agregando assim todo o espaço livre numa única área.

O alocador reduz-se então ao algoritmo A4.4. O algoritmo compactador deve ser tal que, após sua aplicação a estrutura apontada por cada um dos campos de base seja "equivalente" à que o mesmo apontava antes da compactação. Mais exatamente, deve existir uma correspondência biunívoca entre os nós e itens das estruturas originais e os das compactadas, de tal forma que

- (1) nós correspondentes contêm campos correspondentes, na mesma ordem;

áreas LIVRE: endereço {da área livre, suposta no fim do espaço de alocação },

TAMLIV: inteiro de 0 a M;

procedimento ~~ALOCA-NÓS-SEM-LISTA-LIVRE~~:

recebendo T: inteiro de 1 a M,

e área E: campo

efetua

se $T > LIM - LIVRE$, então

~~MARCA-NÓS-ATIVOS~~;

~~COMPACTA-NÓS~~;

se $T > LIM - LIVRE$, então {erro espaço esgotado};

$E \leftarrow LIVRE$;

$LIVRE \leftarrow LIVRE + T$;

Algoritmo A4.4 - Alocador para nós não uniformes, com espaço livre. O procedimento COMPACTA-NÓS pode ser qualquer dos descritos a seguir.

- (2) itens atômicos correspondentes contendam o mesmo conteúdo;
- (3) apontadores correspondentes designem nós correspondentes;
- (4) os nós apontados por um campo da base, antes e depois da compactação, sejam correspondentes.

A compactação envolve portanto duas operações: o deslocamento dos nós, que copia o conteúdo de cada nó das estruturas originais na nova área que lhe corresponde; e a atualização de cada apontador desses nós e da base, de modo que o mesmo passe a designar a nova posição do nó originalmente apontado. Essas duas operações podem ser efetuadas em duas fases distintas, ou simultaneamente.

Como pode ser intuitivo, a compactação é um processo relativamente demorado e complexo, de modo que, em algumas implementações com nós de tamanho variado, pode ser mais conveniente usar a gerência por lista livre, reservando-se a compactação para quando aquela falhar. Em tal caso, os elementos da lista livre devem ser tratados pelo compactador como nós inativos.

Os algoritmos compactadores podem ser úteis mesmo quando os nós tem tamanho uniforme, para, por exemplo, abrir espaço para a pilha de alocação, em sistemas que a utilizam.

Outra função importante dos algoritmos compactadores é diminuir o número de falhas de páginas em sistemas com memória virtual. Os algoritmos de alocação e coleta por lista livre tendem a distribuir aleatoriamente os nós ativos, enquanto que, nos que usam compactação, nós relacionados tendem a estar na mesma página. Este fato tende a diminuir o

número de falhas causadas pelo programa do usuário, ao seguir os apontadores das estruturas.

4.4 Compactação por Tabela de Relocação

Para a atualização dos apontadores, o método mais simples consiste em construir uma tabela, numa área separada da memória, contendo, para cada nó atingível, seu endereço original e o de sua nova área.

Numa primeira fase, um percurso sequencial do espaço de alocação permite deslocar todos os nós marcados para posições consecutivas no início da memória. Assim que um nó é deslocado, é construída a entrada correspondente da tabela.

Numa segunda fase, percorrem-se os campos de base e os dos nós compactados. Cada apontador encontrado é localizado na tabela, obtendo-se desta o novo endereço do nó designado. Este endereço substitui então o apontador original.

Vamos definir um bloco como sendo uma área maximal, formada pela reunião de nós ativos, precedida por uma área inativa do espaço de alocação (considerando-se eventuais elementos da lista livre como áreas inativas). Definamos também um interstício como sendo uma área inativa maximal do espaço de alocação.

Denotaremos por B o número de blocos. Note-se que pode haver nós que não fazem parte de nenhum bloco; isso ocorre quando os primeiros nós do espaço de alocação forem marcados. Denotaremos por N_{AD} o número de nós (e por C_{AD} o número de campos) situados em blocos. Apenas esses N_{AD} nós são realmente deslocados na compactação.

modo entrada: [end: endereço de $n\bar{o}$,
 des: inteiro de 0 a $M-1$];
 procedimento COMPACTADOR-COM-TABELA:
 efetua

1	áreas NB: inteiro de 0 a \bar{B} , TAB: sequência de \bar{B} entradas; E: apontador, MANT: marca-de- $n\bar{o}$; {constroi tabela e compacta} NB \leftarrow 0; MANT \leftarrow 'útil'; E \leftarrow INI; LIVRE \leftarrow INI; repete, enquanto E < LIM,
N	se E \downarrow .marca é 'útil', então
N _A	se MANT é 'inútil', então
B	NB \leftarrow NB + 1; TAB[NB] \leftarrow [end: E; des: E-LIVRE]; MANT \leftarrow 'útil';
N _A	move E \downarrow para LIVRE \downarrow ; LIVRE \leftarrow LIVRE + tam(E \downarrow)
N-N _A	senão
N	MANT \leftarrow 'inútil'; E \leftarrow E + tam(E \downarrow);

Algoritmo A4.5 - Compactador com tabela de relocação. A invocação do procedimento BUSCA localiza J tal que ($J = 0$ ou $TAB[J].end < A$) e ($J = NB$ ou $TAB[J+1].end > A$), usando um método eficiente (por exemplo, busca binária). Os parâmetros \bar{B} , B e N_{AD} são descritos no texto.

	{atualiza apontadores}
	procedimento ATUALIZA:
	recebendo área A: campo,
	efetua
$C_A + C_B$	se A é apontador, então
$P_A + P_B$	área J: inteiro de 0 a NB;
	BUSCA(A, TAB, NB, J);
	se J > 0, então
$P_{AJ} + P_{BJ}$	A ← A - TAB[J].des;
1	repete, para J desde 1 até C_B ,
C_B	ATUALIZA(R[J]);
1	E ← INI;
	repete, enquanto E < LIVRE,
N_A	repete, para J desde 1 até E+.c,
C_A	ATUALIZA(E+.corpo[J]);
N_A	E ← E + tam(E+);

Algoritmo A4.5 - (Continuação)

O deslocamento é em geral efetuado de forma a conservar as posições relativas dos nós pertencentes a um mesmo bloco (na verdade, isto é indispensável no caso de estruturas com nós superpostos). Isto significa que todos os nós de um bloco sofrem o mesmo deslocamento; portanto, é suficiente guardar na tabela o endereço de cada bloco e seu deslocamento. O algoritmo A4.5 usa esse princípio.

O número B de blocos é limitado por $\bar{B} = \min(N_A, N - N_A)$, e este é o número de entradas a reservar na tabela TAB .

O tempo exigido por esse algoritmo é pelo menos $O((P_A + P_B)\log B)$, devido às $(P_A + P_B)$ consultas à tabela TAB , cada uma das quais exige tempo no mínimo $O(\log B)$.

Os símbolos P_{AJ} e P_{BJ} denotam o número de apontadores ativos e de base, resp., que precisam ser ajustados (isto é, que apontam para nós realmente deslocados).

Os endereços E e $LIVRE$ percorrem sequencialmente a memória, de modo que o número de falhas de páginas causadas pelo algoritmo pode ser estimado em $G + 2G_{AC}$ (G_{AC} é o número de páginas ativas após a compactação). A elas devem ser adicionadas as falhas geradas pela etapa preliminar de marcação, que são no mínimo $P_B + P_A$.

Note que referências à tabela TAB não foram contadas como falhas de páginas.

4.5 Compactação com Tabela nos Interstícios

A tabela do algoritmo A4.5 pode ser armazenada nos interstícios entre os blocos. Essa idéia foi proposta por B. K. Haddon e W. M. Waite em {Had/Wai 6/67}, e nela baseia-se o algoritmo A4.6.

procedimento COMPACTADOR-COM-TABELA-NOS-INTERSTÍCIOS:
efetua

```

1      áreas NB: inteiro de 0 a B, inicialmente 0,
      INITB: endereço de sequência de NB entradas,
      FIMTB, PITB, IDTB, PFTB: endereços de entradas,
      PINA, PATV, QINA, QATV, FMOV, E: apontadores,
      COMP, COMPTB: inteiros de 1 a M;

      {primeira fase: deslocamento dos nós e construção da tabela}

      PINA ← INI; PATV ← INI; INITB ← INI; FIMTB ← INI;
      MALHA: repete indefinidamente
      B*+1      QATV ← PATV;
      repete
      B+1      {determina próximo interstício (QINA↑)}
      QINA ← QATV;
      repete,
      NA+B+1      enquanto QINA < LIVRE e QINA↑.marca for 'útil',
      NA          QINA ← QINA + tam(QINA↑);
      {determina bloco seguinte (QATV↑)}
      B+1      QATV ← QINA;
      repete,
      N-NA+B+1      enquanto QATV < LIVRE e QATV↑.marca for 'inútil',
      N-NA          QATV ← QATV + tam(QATV↑);
      {determina futura posição da tabela de relocação}
      B+1      COMP ← QINA - PATV; FMOV ← PINA - COMP;
      PITB ← INITB; COMPTB ← 0;
      repete,
      K2+B+1      enquanto PITB < FMOV,
      K2          [PITB ← PITB + tam(entrada);
                  COMPTB ← COMPTB + tam(entrada);
      B+1      se PITB > FIMTB, então
      K1          [PITB ← FMOV; COMPTB ← FIMTB - INITB;
                  IDTB ← PITB
      senão
      B+1+K1      [IDTB ← FIMTB;
      B+1      [PFTB ← IDTB + COMPTB;

      até que QATV > LIVRE ou QATV - PFTB > tam(entrada);
  
```

Algoritmo A4.6 - Compactador com tabela nos interstícios.
(continua)


```

B*      {inclui nova entrada na tabela}
        FIMTB ← [end: QATV, des: QATV - FMOV];
        NB ← NB + 1; FIMTB ← FIMTB + tam(entrada);
        prepara proxima iteração
        PATV ← QATV; PINA ← FMOV;

1      LIVRE ← FMOV;
        {segunda fase: atualização dos apontadores}
        ORDENA-TABELA(INITB, NB);
        procedimento ATUALIZA:
            recebendo área A: campo,
            efetua
                [ se A é apontador, então
                    [ área J: inteiro de 0 a NB;
                        BUSCA(A, INITB, NB, J);
                        se J > 0, então
                            A ← A - INITB[J].des;
                ]
            ]
        repete, para J desde 1 até CB,
            ATUALIZA(R[J]);

1      E ← INI;
        repete,
            enquanto E ← LIVRE,
                [ repete, para J desde 1 até E+.c,
                    ATUALIZA(E+.corpo[J]);
                    E ← E + tam(E+)
                ]

```

Algoritmo A4.6 (conclusão)

O procedimento ORDENA-TABELA ordena a tabela de relocação INITB segundo o item end. O procedimento BUSCA é o mesmo de A4.5. Os parâmetros K_1 , K_2 e B^* são discutidos no texto.

Nesse algoritmo, como em A4.5, os blocos são deslocados sequencialmente, em ordem crescente de endereços. A tabela de relocação é armazenada no espaço existente entre a cópia do último nó deslocado, e o primeiro nó ainda por deslocar. Este espaço corresponde à aglutinação dos interstícios encontrados até o momento.

Quando um novo bloco é deslocado, a tabela de relocação (ou uma parte dela) deve ser movida para a frente, para abrir espaço para o mesmo, e uma nova entrada lhe deve ser acrescentada.

Para que isto seja possível, o espaço livre já acumulado, mais o interstício que segue o bloco a deslocar, devem ser suficientes para a tabela assim aumentada. Se isto não ocorrer, é necessário desprezar o próximo interstício, tratando-o como se fosse uma área ativa, e repetir a tentativa com o interstício seguinte.

Note-se a maneira curiosa com que a tabela se desloca (fig. F4.1): a cada etapa, apenas a parte que seria recoberta pelo bloco a deslocar é movida para a frente. Isso destrói a ordem da tabela, que é reordenada novamente antes da atualização, em tempo $O(B_* \log B_*)$.

A ordem da tabela seria preservada se toda ela fosse deslocada a cada passo. Entretanto, desta forma o tempo gasto nesses deslocamentos poderia ser, em certas situações, $O(M_A^2)$. O algoritmo A4.6, entretanto, nunca move duas entradas da tabela para o mesmo lugar na memória, de modo que o tempo gasto em deslocar a tabela é no máximo $O(M_A)$.

O número B_* de "blocos" deslocados é menor ou igual ao número B de blocos realmente existentes, e só é menor se o tamanho médio dos k primeiros interstícios (para algum k) for menor que o tamanho de uma entrada da tabela.

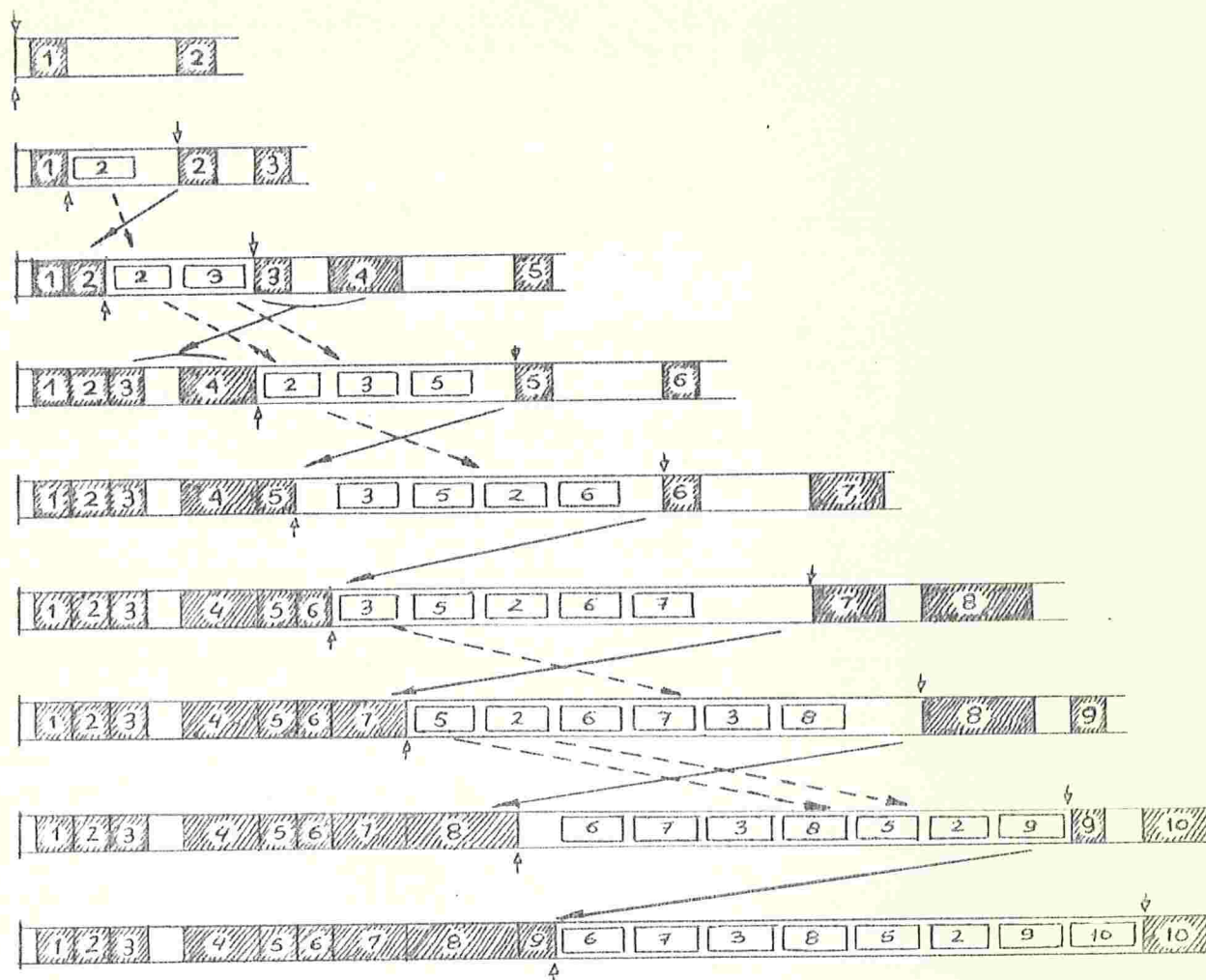


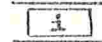
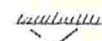


Figura F4.1 -Ilustração da fase de compactação do algoritmo A4.6. Em cada linha está representada a situação no início de uma iteração da malha mais externa.

Legenda: \uparrow, \downarrow = posição dos apontadores PINA e PATV.

 = i-ésimo bloco,  = interstício.

 = entrada da tabela correspondente ao i-ésimo bloco.

 = movimentação de blocos (—) e de entradas da tabela (---).

la. Sô neste caso o algoritmo é obrigado a deslocar vários blocos, e os interstícios entre eles, como se fossem um único bloco.

Os parâmetros K_1 e K_2 não são determináveis facilmente; mas K_1 é no máximo $B+1$, e K_2 é no máximo $M_A/\text{tam}(\text{entrada})+K_1$.

Se supusermos que a memória rápida é suficiente para conter a tabela completa, e mais algumas páginas, o número de falhas causadas por esse algoritmo pode ser estimado em $G + 2G_{AC}$, como em A4.5. Se a memória rápida não for suficiente para conter a tabela, termos aproximadamente G_{AC} falhas adicionais para sua construção, e mais um número considerável para a ordenação e as buscas. Novamente, não consideramos as P_A+P_B falhas (no mínimo) geradas pela marcação.

4.6 Compactação de Nós Uniformes

Para a compactação de estruturas com nós de tamanho uniforme, existe um algoritmo bastante simples e eficiente, desenvolvido originalmente para a linguagem LISP 1.5. Esse algoritmo foi descrito por Knuth {Knu 68 - 2.3.5 ex. 9}, e atribuído por ele a Daniel J. Edwards.

O algoritmo de Edwards (A4.7) transfere o nó ativo de maior endereço para o nó inativo de menor endereço, repetindo essa operação até que todos os nós ativos estejam em posições consecutivas.

Desta forma, as áreas originais dos nós deslocados não são preenchidas com outros nós, e são usadas pelo algoritmo para construir a tabela de relocação. Para tanto, o en

procedimento COMPACTADOR-DE-EDWARDS:
efetua

	<pre> área E: apontador; {deslocamento dos nós} LIVRE ← INI; E ← LIM; MALHA: repete indefinidamente repete, enquanto LIVRE↑.marca é 'útil', LIVRE↑.marca ← 'inútil'; LIVRE ← LIVRE + tam(nó); se LIVRE ≥ E, então termina MALHA; repete, E ← E - tam(nó); se E ≤ LIVRE, então termina MALHA; até que E↑.marca seja 'inútil'; move E↑ para LIVRE↑; E↑.marca ← 'inútil'; move LIVRE para E↑.corpo; LIVRE ← LIVRE + tam(nó); {atualização dos apontadores} procedimento ATUALIZA: recebendo área A: campo, efetua se A é apontador, e A ≥ LIVRE, então A ← A↑.corpo como apontador; LIM ← LIVRE; repete, para J desde 1 até C_B, ATUALIZA(R[J]); E ← INI; repete, enquanto E < LIM, repete, para J desde 1 até Ĉ, ATUALIZA(E↑.corpo[J]); E ← E + tam(nó) </pre>
1	
$N_{AD}+1$	
$N_{AD}+1-K_1$	
N_A-N_{AD}	
K_1	
$N_{AD}+1-K_1$	
$N-N_A+1-K_1$	
$1-K_1$	
$N-N_A$	
N_{AD}	
C_A+C_B	
P_A+P_B	
P_A+P_B	
$P_{AJ}+P_{BJ}$	
1	
C_B	
1	
N_A	
C_A	
N_A	

Algoritmo A4.7 - O compactador de Edwards para nós de tamanho uniforme. O parâmetro K_1 pode ser 0 ou 1.

dereço da nova posição de cada nó deslocado é armazenado no primeiro campo de sua área original. Isto permite atualizar eficientemente todos os apontadores para esse nó.

Graças a essa idéia, o algoritmo de Edwards exige tempo linear em N , e não precisa de nenhum espaço auxiliar (além das marcas de acessibilidade).

A área original de um nó deslocado daremos o nome de nó fantasma. Se substituirmos idealmente todo apontador para um nó fantasma pelo primeiro campo deste, podemos afirmar que a estrutura apontada por um campo de base permanece equivalente à apontada antes da compactação, durante toda a execução do algoritmo A4.7.

O conceito de "nó fantasma", e a propriedade invariante acima, ocorrem em muitos algoritmos de compactação e de recuperação por cópia (cap. V).

Os parâmetros N_{AD} , P_{AJ} e P_{BJ} (número de nós deslocados, e de apontadores ativos e de base que precisam ser ajustados) são menores para este algoritmo do que para os anteriores. Se os nós estiverem aleatoriamente distribuídos, os valores médios dos mesmos serão $N_A(N_X/N)$, $P_A(N_X/N)$ e $P_B(N_X/N)$, onde N_X é o número de nós flutuantes inativos. Para os algoritmos anteriores, esses parâmetros seriam praticamente N_A , P_A e P_B .

Note-se que as operações $E \leftarrow E - \text{tam}(\text{nó})$ e $\text{LIVRE} \leftarrow \text{LIVRE} + \text{tam}(\text{nó})$ são efetuadas, no total, N vezes. O tempo gasto nelas tende a dominar toda a recuperação, se N_A for muito menor que N , pois a marcação e as demais operações de A4.7 exigem tempo no máximo $O(M_A + M_B)$.

O número de falhas de páginas causadas por A4.7 pode ser estimado em $G + G_{AC} + (P_{AJ} + P_{BJ})$.

4.7 Compactação de Estruturas com Poucos Nós Ativos

Ainda para estruturas com nós de tamanho uniforme, e-xiste um algoritmo compactador (A4.8) cujo tempo de execução é no máximo $O(M_A + M_B)$, isto é, não depende do número de nós inativos.

Como no algoritmo de Edwards, o endereço para onde cada nó foi deslocado é armazenado no primeiro campo do nó original. A rotina de marcação deve determinar o número N_A de nós ativos, a partir do qual é calculado o parâmetro SEP.

O algoritmo A4.8 procura preencher todos os nós marcados entre $INI\downarrow$ (inclusive) e $SEP\downarrow$ (exclusive) com cópias dos nós atingíveis situados além de $SEP\downarrow$.

Primeiramente, são copiados os nós nestas condições que são apontados por um campo de base, ou por algum nó marcado aquém de $SEP\downarrow$. Em seguida, essas cópias são examinadas, deslocando-se seus filhos se for o caso. Esse exame é repetido para estes filhos, para os filhos destes filhos, e assim por diante, até que todos os nós além de LIM tenham sido copiados.

Note-se que no SEGUNDO PASSO, E é sempre menor que LIVRE, enquanto ainda houver nós a copiar. De fato, quando $E = LIVRE$, tem-se que todo campo de base, todo nó marcado entre $INI\downarrow$ e $SEP\downarrow$, e todo nó não marcado entre $SEP\downarrow$ e $LIVRE\downarrow$ apontam para nós marcados entre $INI\downarrow$ e $SEP\downarrow$, ou nós não marcados entre $INI\downarrow$ e $LIVRE\downarrow$. Todo nó atingível estará então entre $INI\downarrow$ e $SEP\downarrow$; como há exatamente N_A nós nesse intervalo, conclui-se que $LIVRE = SEP$ e todos os nós marcados além de $SEP\downarrow$ já foram deslocados.

A movimentação dos nós ativos consome tempo $O(M_A)$ no máximo, e o mesmo pode ser dito das demais (exceto as que

modos $\bar{n}\bar{o}$: [marca: marca-de- $\bar{n}\bar{o}$
 corpo: sequência de \bar{c} campos];
 campo: união uniforme disjunta de (\bar{a} tomo, apontador);
 procedimento COMPACTA-NÓS-UNIFORMES:
 efetua
 1 [\bar{a} reas E, SEP: apontadores
 procedimento DESLOCA-E-ATUALIZA:
 recebendo \bar{a} rea A: campo
 efetua
 se A \bar{e} apontador,
 e $A \geq \text{SEP}$, então
 se $A\downarrow.\text{marca}$ \bar{e} 'útil', então
 [$A\downarrow.\text{marca} \leftarrow \text{'inútil'}$;
 move $A\downarrow$ para LIVRE ;
 move LIVRE para $A\downarrow.\text{corpo}$;
 repete
 LIVRE $\leftarrow \text{LIVRE} + \text{tam}(\bar{n}\bar{o})$
 atê que $\text{LIVRE}\downarrow.\text{marca}$ seja 'inútil';
 $A \leftarrow A\downarrow.\text{corpo}$ como apontador;
]

$C_A + C_B$
 $P_A + P_B$
 $P_{AJ} + P_{BJ}$
 N_{AD}

 $N_A - K_1$

 $P_{AJ} + P_{BJ}$

Algoritmo A4.8 - Compactação de nós uniformes em tempo $O(M_A)$. O parâmetro K_1 é no máximo N_A , e não influi no tempo total de execução. (Continua)

1	LIVRE \leftarrow INI; SEP \leftarrow INI + tam(nō) \times N _A ;
	repete,
K ₁ +1	enquanto LIVRE↓.marca \bar{e} 'útil',
K ₁	LIVRE \leftarrow LIVRE + tam(nō);
	PRIMEIRO PASSO:
1	[repete, para J desde 1 até C _B ,
C _B	DESLOCA-E-ATUALIZA(R[J]);
1	E \leftarrow INI;
	repete,
N _A +1	enquanto E < SEP,
N _A	[se E↓.marca \bar{e} 'útil', então
N _A -N _{AD}	repete, para J desde 1 até \bar{c} ,
C _A -C _{AD}	DESLOCA-E-ATUALIZA(E↓.corpo[J]);
N _A	E \leftarrow E + tam(nō);
	SEGUNDO PASSO:
1	E \leftarrow INI;
	repete,
N _A +1	enquanto E < SEP,
N _A	[se E↓.marca \bar{e} 'inútil', então
N _{AD}	repete, para J desde 1 até \bar{c} ,
C _{AD}	DESLOCA-E-ATUALIZA(E↓.corpo[J]);
	senão
N _A -N _{AD}	E↓.marca \leftarrow 'inútil';
N _A	E \leftarrow E + tam(nō)

Algoritmo A4.8 (Continuação)

envolvem os campos de base). Pode-se concluir então que o tempo total é no máximo $O(M_A + M_B)$.

Comparando-se o número de vezes que cada operação é efetuada em A4.7 e A4.8, conclui-se que as diferenças significativas ocorrem nas operações de incremento/decremento de endereços ($N + N_A$ vezes em A4.7, contra $3N_A$ em A4.8), comparação de endereços ($N + N_A + P_A + P_B$ contra $2N_A + P_A + P_B$), e teste das marcas dos nós (N contra $3N_A + P_{AJ} + P_{BJ}$). Para N_A/N suficientemente baixo (pouco menor que $1/2$) o algoritmo A4.8 será mais eficiente que o de Edwards.

O número de falhas causadas por este algoritmo é $3G_{AC} + P_{AJ} + P_{BJ}$, que é menor que o de A4.7 quando $M_A/M < 1/2$.

4.8 Compactador Genérico Sem Tabela

Um algoritmo recente, devido a F.L.Morris (Mor 8/78) permite compactar estruturas genéricas (com nós de tamanhos variados) em tempo no máximo $O(M_A + M_B)$ usando apenas um bit adicional por campo. Compare-se isto com o algoritmo A4.6, que não precisa desse bit mas exige tempo pelo menos $O((P_A + P_B) \log B)$.

A técnica usada pelo algoritmo de Morris consiste em formar, para cada nó atingível, uma lista ligada com todos os campos apontadores que o designam. (Fig. F4.2). O endereço do primeiro elemento dessa lista é armazenado no próprio nó apontado, e o conteúdo deste é temporariamente salvo no último dos elementos da lista, em lugar do campo atômico que lá deveria haver. Essa lista é a dita a lista invertida do nó.

Para construí-la, o algoritmo percorre sequencialmen

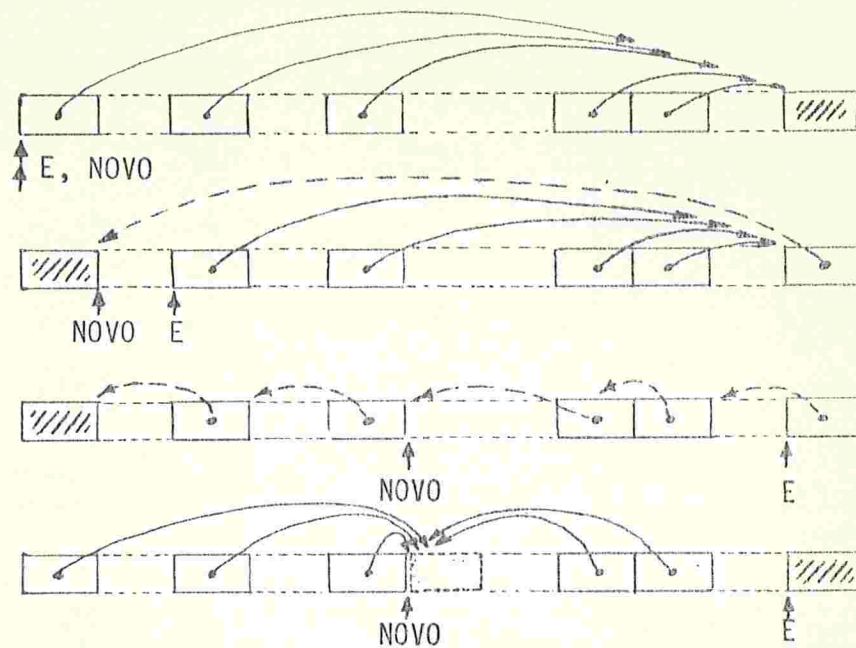


Figura F4.2 - Construção e atualização da lista invertida de um único campo, como descrita por Fischer. NOVO indica a futura posição do nó designado por E.

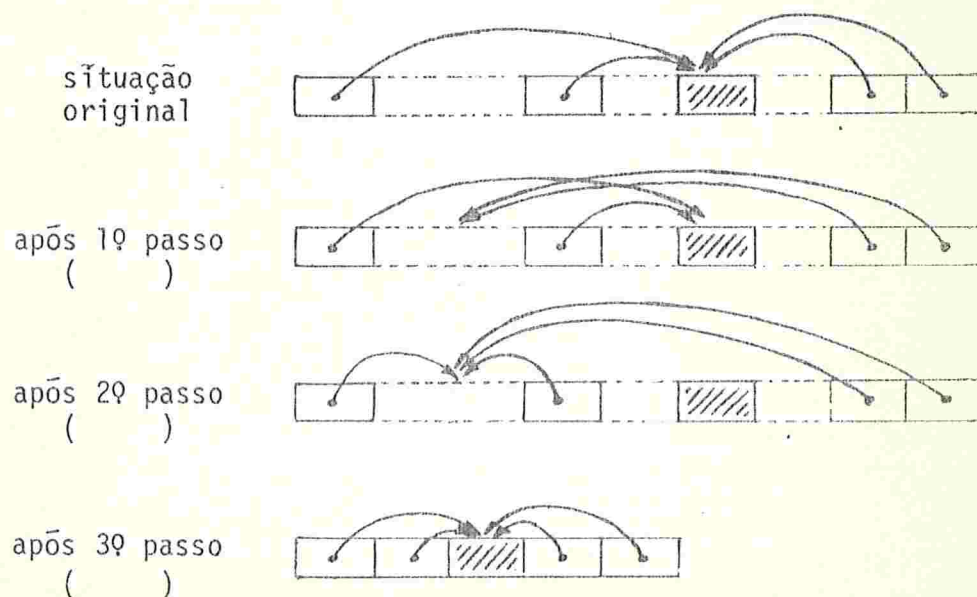


Figura F4.3 - Os tres passos do algoritmo de Morris (A4.9), e seu efeito sobre a estrutura indicada na figura.

te o espaço de alocação, e adiciona cada apontador encontrado à lista invertida do nó designado por ele.

Quando um nó é encontrado, o algoritmo já conhece o endereço que o mesmo ocupará após o deslocamento, tendo somado os tamanhos de todos os nós marcados que encontrou até o momento.

A lista invertida permite então localizar facilmente todos os campos que designavam o nó, e atualizá-los com o futuro endereço do mesmo. Ao fim dessa operação, o conteúdo o original do nó é restaurado, e o percurso continua.

Esta técnica foi proposta originalmente por D.A.Fisher {Fis 7/74} que se limitou a estruturas coerentes, em que todo nó tem endereço menor que o de seus filhos. Essa restrição é importante, pois garante que todos os campos apontadores serão percorridos antes dos nós que eles referenciam, e permite distinguir, pelo seu "sentido", os apontadores de listas invertidas dos apontadores originais.

Para aplicar essa técnica a estruturas genéricas, o algoritmo de Morris utiliza um bit adicional em cada campo apontador, para distinguir os apontadores invertidos, e efetua dois percursos sequenciais, em sentidos opostos, através do espaço de alocação, tratando em cada um apenas dos apontadores com "sentido" conveniente. (Fig. F4.3).

O deslocamento dos nós para suas posições finais é feito por um terceiro passo, após a atualização de todos os apontadores. Este passo pode ser combinado com o segundo, desde que os sentidos concordem (crescente se a compactação for na direção de início da memória, e decrescente em caso contrário).

É importante neste algoritmo que cada nó tenha espa-

ção suficiente para armazenar um endereço (o de sua lista invertida). Outra necessidade do algoritmo é que seja possível percorrer sequencialmente os nós nos dois sentidos. Isto significa que, num dos passos, o tamanho de cada nó deve ser determinado partindo-se apenas do endereço do nó seguinte.

Como esta exigência complica bastante a descrição do algoritmo, suporemos que os nós são amorfos, e que os campos são auto-descritos e de tamanho uniforme. Desta forma, os percursos podem ser efetuados facilmente, campo-a-campo em vez de nó-a-nó (algoritmo A4.9).

Suporemos uma marca em cada campo, que deve ser definida por algum algoritmo de marcação apropriado. Esse algoritmo deve calcular também o número C_A de campos ativos.

Cada campo apontador contém um item adicional, inv, para distinguir apontadores invertidos.

Os campos de base são incluídos nas listas invertidas no primeiro passo, e atualizados nessa etapa. Apontadores que designam a si próprios são também tratados, separadamente, no primeiro passo.

Note-se que o sentido do primeiro passo deve concordar com o sentido do deslocamento dos nós. Caso contrário, um apontador atualizado no primeiro passo pode ter seu sentido invertido, e ser indevidamente atualizado também no segundo.

Este algoritmo gera aproximadamente $2G + G_{AC} + P_B + 2P_A$ falhas de páginas. O termo $2P_A$ deve-se ao fato de cada apontador ser seguido duas vezes: na inversão e na atualização.

modos apontador: [c: inteiro de 1 a \hat{c} ,
 inv: união uniforme disjunta de
 ('invertido', 'normal'),
 ender: endereço de sequência de c campos],
 campo: [marca: ~~marca-de-não~~,
 corpo: união uniforme disjunta de
 (átomo, apontador)];

procedimento COMPACTADOR-MORRIS:

efetua

1

áreas E, NOVO: endereços de nós;

procedimento INVERTE:

usando T: campo,

efetua

$P_B + P_{A<} + P_{A>}$

áreas A: campo, R: endereço de campo;

$R \leftarrow T.corpo.ender;$

$A.corpo \leftarrow R\downarrow.corpo;$

$R\downarrow.corpo \leftarrow [c: T.corpo.c,$
 inv: 'invertido',
 ender: end(T)];

$T.corpo \leftarrow A.corpo$

procedimento ATUALIZA-LISTA:

usando A: campo,

efetua

$2C_A$

áreas T: campo, R: endereço de campo;

repete,

enquanto A.corpo for apontador,
 e A.corpo.inv for 'invertido',

$2C_A + P_B + P_{A<} + P_{A>}$

$3P_{A<} + P_{A>}$

$P_B + P_A + P_A$

$R \leftarrow A.corpo.ender;$
 $T.corpo \leftarrow R\downarrow.corpo;$
 $R\downarrow.corpo \leftarrow [c: A.corpo.c,$
 inv: 'normal',
 ender: NOVO];

$A.corpo \leftarrow T.corpo$

Algoritmo A4.9 - Compactador de Morris para nós amorfos com campos auto-descritos e de tamanho uniforme. C_{AD} é o número de campos realmente deslocados; $P_{A<}$, $P_{A>}$ e $P_{A=}$ são o número de apontadores acessíveis que precedidos por, seguidos por, ou coincidentes com os campos por eles designados.

(Continua)

1	PRIMEIRO-PASSO:
C_B	repete, para J desde 1 até C_B ,
P_B	se $R[J].\text{corpo}$ for apontador, então
1	$\text{INVERTE}(R[J])$;
	$E \leftarrow \text{LIVRE}$; $\text{NOVO} \leftarrow \text{INI} + C_A \cdot \text{tam}(\text{campo})$;
$C+1$	repete
C	enquanto $E > \text{INI}$,
C_A	$E \leftarrow E - \text{tam}(\text{campo})$;
P_A	se $E \downarrow.\text{marca}$ é 'útil', então
$P_{A=}$	$\text{NOVO} \leftarrow \text{NOVO} - \text{tam}(\text{campo})$;
$P_{A>} + P_{A<}$	$\text{ATUALIZA-LISTA}(E \downarrow)$;
$P_{A<}$	se $E \downarrow.\text{corpo}$ for apontador, então
	se $E \downarrow.\text{corpo.ender} = E$, então
	$E \downarrow.\text{corpo.ender} \leftarrow \text{NOVO}$
	senão
	se $E \downarrow.\text{corpo.ender} < E$, então
	$\text{INVERTE}(E \downarrow)$
1	SEGUNDO-E-TERCEIRO-PASSOS:
$C+1$	$E \leftarrow \text{INI}$; $\text{NOVO} \leftarrow \text{INI}$;
C	repete,
C_A	enquanto $E < \text{LIVRE}$,
P_A	se $E \downarrow.\text{marca}$ é 'útil', então
$P_{A>}$	$\text{ATUALIZA-LISTA}(E \downarrow)$;
	se $E \downarrow.\text{corpo}$ for apontador,
	e se $E \downarrow.\text{corpo.ender} > E$, então
	$\text{INVERTE}(E \downarrow)$;
C_{AD}	se $E \neq \text{NOVO}$, então
C_A	move $E \downarrow$ para $\text{NOVO} \downarrow$;
C	$E \downarrow.\text{marca} \leftarrow \text{'inútil'}$;
1	$\text{NOVO} \leftarrow \text{NOVO} + \text{tam}(\text{campo})$;
	$E \leftarrow E + \text{tam}(\text{campo})$;
	$\text{LIVRE} \leftarrow \text{NOVO}$;

Algoritmo A4.9 - (Continuação)

4.9 Compactação com Nós e Campos Amorfos

Estruturas com nós amorfos e sobrepostos, mas com campos auto-descritos, não apresentam grandes dificuldades aos compactadores genéricos A4.5, A4.6 e A4.9, bastando que cada campo seja individualmente marcado e que os percursos sejam feitos campo-a-campo em vez de nó-a-nó. Como esses algoritmos preservam a ordem relativa dos elementos deslocados, campos ativos adjacentes continuarão adjacentes após a compactação, preservando-se assim a integridade dos nós.

Se os campos forem amorfos e de tamanhos variados, surgem inúmeras dificuldades. Torna-se impossível percorrer sequencialmente os campos, e distinguir apontadores de átomos; e o uso de uma marca em cada campo, no caso de campos muito pequenos, pode ser um desperdício considerável de espaço.

Os algoritmos de coleta sem compactação tem o mesmo problema de percurso, em estruturas com campos amorfos. Além disso, campos de tamanho pequeno tendem a produzir interstícios pequenos demais para serem incluídos na lista livre.

A técnica de marcação por segmentos, vista no capítulo III, pode ser usada com relativo sucesso também na coleta. Em vez de nós ou campos, o objetivo passa a ser coletar os segmentos inativos, ou compactar os ativos (preservando, naturalmente, a ordem relativa dos mesmos).

Se não for usada a compactação, o único cuidado especial a tomar é garantir que o tamanho de um segmento seja suficiente para sua inclusão na lista livre (isto é, deve ser capaz de conter o tamanho do elemento a que pertence, e a ligação para o próximo).

Para a compactação, são necessárias precauções mais elaboradas. A atualização dos apontadores exige que os mesmos possam ser localizados durante o percurso linear dos segmentos. Para que isso seja possível, é necessário obrigar cada apontador a ocupar uma posição fixa em relação ao início de um segmento (seu segmento principal), e exigir que a rotina de marcação indique, por meio de uma segunda marca em cada um, os segmentos principais dos apontadores ativos.

Com esta modificação, os compactadores por tabela (A4.5 e A4.6) podem ser aplicados sem problemas. O algoritmo de Morris (A4.9), porém, apresenta uma dificuldade adicional: um campo pode ser pequeno demais para conter o endereço de sua lista invertida.

A solução para este problema é armazenar em cada apontador, em vez do endereço do campo designado, um par $[\text{endseg}, \text{desloc}]$, onde endseg é o endereço do segmento onde começa o campo, e desloc a diferença entre o endereço deste último e endseg .

Como a compactação é efetuada ao nível de segmentos, na atualização dos apontadores é necessário modificar apenas o item endseg . Mais ainda, todos os apontadores com mesmo endseg sofrerão a mesma atualização, independentemente de seu desloc .

Para aplicar o algoritmo A4.9, então, deve-se obrigar o item endseg a estar contido integralmente no segmento principal do apontador. Cada lista invertida é formada então pelos segmentos principais dos apontadores que designam um mesmo segmento, ligados entre si por seus endsgs . O endereço do primeiro desses segmentos é guardado no endseg do apontado, e o conteúdo original deste é, como dantes, salvo no endseg do último elemento da lista. (Fig. F4.4).

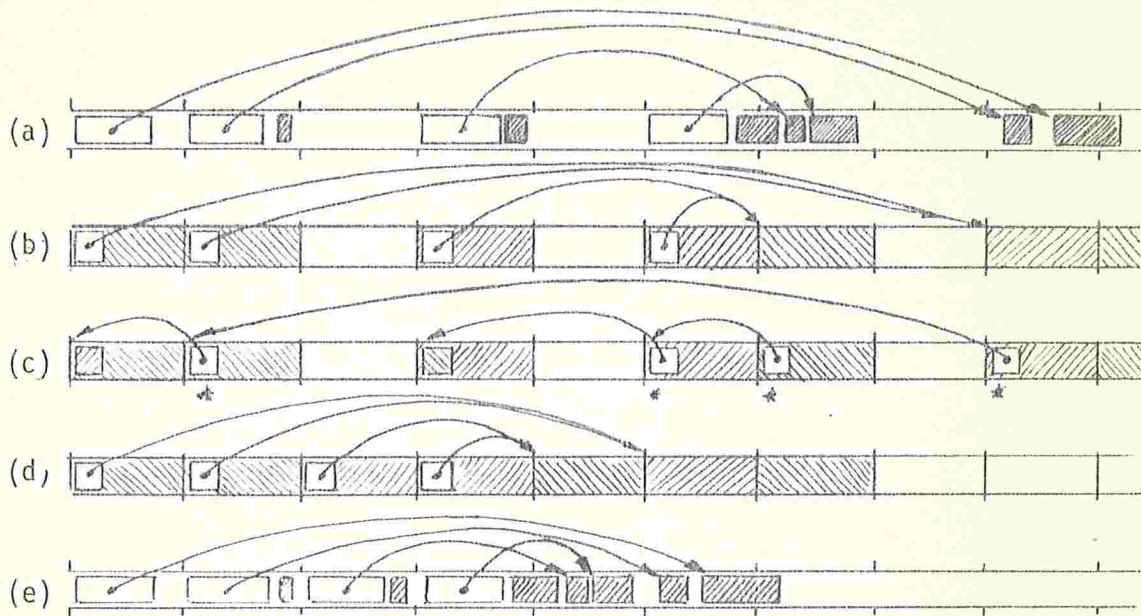


Figura F4.4 - Ilustração do funcionamento do algoritmo A4.9 com segmentação do espaço de alocação.

Legenda: (a) situação original:

▨ = átomos, □ = apontadores,
— = fronteiras de segmentos;

(b) mesma situação, considerada a nível de segmentos:

▨ = segmento ativo sem apontadores,
□ = segmento inativo,
▨ = segmento ativo com apontador (principal),
□ = item endseg do segmento principal;

(c) idem, após a construção das listas invertidas,

(d) idem, após a compactação,

(e) idem (d), novamente ao nível de campos, como em (a).

Durante a compactação, é necessário distinguir quatro categorias de segmentos: os inativos; os ativos, que não contem apontadores; os que contem apontadores ativos (segmentos principais); e os que contem apontadores invertidos. Essa indicação pode ser feita por par de bits em cada segmento (de modo que o algoritmo A4.9 não exige neste caso mais espaço que A4.5 e A4.6), inicializados pela rotina de marcação.

4.10 O Compactador de Zave

O compactador descrito por D.A Zave em {Zav 7/75} é usado após a marcação pelo algoritmo especializado, A3.10. Recordemos que este algoritmo, em vez de marcar os nós ou campos ativos, forma uma lista ligada com todos os apontadores acessíveis e os de base, encadeados por um item adicional dos mesmos.

O algoritmo compactador A4.10 ordena os apontadores que ocorrem nessa lista, segundo os endereços dos nós apontados. Um percurso sequencial da lista ordenada permite ao mesmo tempo atualizar os apontadores e deslocar os nós para suas posições definitivas.

A ordenação da lista de apontadores pode ser efetuada, por exemplo, pelo método de classificação repetida ("radix sort"; vide {Knu 73 - 5.2.5}), e exige tempo pelo menos $O((P_A + P_B) \log(P_A + P_B))$.

A atualização é compactação propriamente ditas exigem tempo $O(M_A) + (P_A + P_B)$. O parâmetro K_2 é mais difícil de quantificar, mas de qualquer forma é no máximo $P_A + P_B$. O comprimento total das áreas deslocadas pelo comando mova é M_A .

modos ligação: união uniforme disjunta de ('nada', endereço de apontador),

campo: união disjunta de (átomo, apontador),

apontador: [c: inteiro de 1 a \bar{c} ,
 en: endereço de sequência de c campos,
 marca: ~~marca-de-não~~,
 prox: ligação];

procedimento COMPACTADOR-ZAVE:

recebendo área LAP: ligação {para a lista de apontadores ativos},

efetua

```

1  [ áreas NOVO, VELHO, AUX: endereços de campos,
    E: ligação,
    T: inteiro de 1 a M;
    ORDENA-LISTA(LAP);
    NOVO ← INI; VELHO ← INI; E ← LAP;
    repete,
    enquanto E ≠ 'nada',
      [ AUX ← E↓.en + tam(E↓.en↓);
        se E↓.en > VELHO, então
          VELHO ← E↓.en;
          E↓.en ← E↓.en - (VELHO - NOVO);
          se AUX > VELHO, então
            [ T ← AUX - VELHO;
              move VELHO↓ tamanho T para NOVO↓;
              VELHO ← VELHO + T;
              NOVO ← NOVO + T;
            ]
          E ← E↓.prox;
        ]
    ] LIVRE ← NOVO

```

Algoritmo A4.10 - O compactador de Zave. O procedimento ORDENA-LISTA deve rearranjar os elementos da lista ligada apontada por LAP, em ordem crescente segundo o item en. O parâmetro K_1 é no máximo 1; B é o número de blocos no espaço de alocação, e K_2 é no máximo $P_A + P_B$.

Apesar de exposto para estruturas com campos auto-descritos, o compactador de Zave aplica-se sem grandes alterações até mesmo a estruturas com campos amorfos de qualquer tamanho.

Uma característica importante deste algoritmo, que é compartilhada apenas pelo compactador A4.8, é que seu tempo de execução depende apenas do tamanho das estruturas ativas. Embora a ordenação exija pelo menos tempo $O((P_A + P_B) \log(P_A + P_B))$, o efeito da mesma não é muito sensível em situações típicas. Levando-se em conta a característica acima mencionada, conclui-se que este algoritmo pode concorrer seriamente com os demais, especialmente quando P_A/C_A é pequeno e M_A é bem menor que M .

O número de falhas causadas por este compactador é limitado por $(P_A + P_B) + G_{AC} + O((P_A + P_B) \log(P_A + P_B))$, sendo este último termo causado pela ordenação da lista de apontadores.

CAPÍTULO V - RECUPERADORES POR CÓPIA

5.1 Introdução

Os métodos de coleta com lista livre e de compactação vistos no capítulo anterior são justificáveis quando o principal objetivo é aproveitar ao máximo o espaço de alocação.

Em sistemas com memória virtual, este objetivo é em geral secundário, pois o tamanho do espaço de alocação habitualmente é maior que a soma de todos os nós criados no decorrer do processamento. A compactação dos nós ativos é desejável, nesse caso, para concentrar as estruturas em uso num número menor de páginas.

Em princípio, os algoritmos de marcação e compactação do capítulo anterior podem ser usados para este fim. Entretanto, existem algoritmos específicos para este caso, que, aproveitando a disponibilidade virtualmente ilimitada de espaço, são mais simples e podem vir a ser mais eficientes, em tempo e em número de acessos.

Nestes algoritmos, o espaço de alocação é dividido em dois semi-espacos de mesmo tamanho. Entre uma recuperação e outra, as estruturas acessíveis estão integralmente localizadas num desses semi-espacos, e a alocação de novos nós é feita dentro deste. A recuperação consiste em copiar os nós ativos para posições consecutivas no outro semi-espaco, atualizando todos os apontadores de modo que as estruturas resultantes sejam equivalentes às originais. Ao fim da recuperação, o papel dos dois semi-espacos é invertido.

Exceto durante a cópia dos nós, todos os endereços manipulados pelo programa são referentes a áreas de um mesmo semi-espço. Portanto, a mesma representação pode ser usada para um endereço num semi-espço, e para seu "homólogo" no outro, economizando-se desta forma um bit na sua codificação.

A operação $\epsilon\uparrow$ terá interpretação ambígua apenas durante a recuperação, em que áreas em ambos os semi-espços são manipuladas simultaneamente. Nesse caso, convencionaremos que $\epsilon\uparrow$ significa sempre a área de endereço ϵ no semi-espço original (de onde as estruturas estão copiadas). Usaremos $\epsilon\uparrow$ para denotar a área de mesmo endereço situada no semi-espço novo (para onde a cópia está sendo realizada). Desta forma, a troca de semi-espços, ao fim da recuperação, é feita permutando-se o sentido das operações " \uparrow " e " \downarrow ".

Geralmente, nos algoritmos deste capítulo, pode-se dizer "a priori" a que semi-espço um endereço se refere. Quando tal não acontecer, usaremos indicadores booleanos para distinguir os dois casos.

As estruturas originais não são mais necessárias depois de terminada a cópia, de modo que o algoritmo copiador não precisa mantê-las intactas. Na verdade, todos os algoritmos deste capítulo destroem as estruturas originais, durante sua execução. Um nome mais adequado para esta classe de algoritmos seria "recuperadores por movimentação".

O uso das estruturas originais como áreas de trabalho permite obter recuperadores mais eficientes que os algoritmos que efetivamente copiam estruturas.

Na análise da eficiência dos algoritmos de cópia, os parâmetros N , N_A , M , M_A , etc. serão referentes ao semi-espço original.

Neste capítulo, salvo observações em contrário, suporemos que as estruturas tem nós disjuntos e auto-descritos, com o formato abaixo,

```
modo nō: [marca: marca-de-nō,
          c: inteiro de 1 a ĉ,
          corpo: sequência de c campos]
```

e que o tamanho de um nō é suficiente para conter uma marca-de-nō e um endereço.

5.2 Cópia com Marcação Prévia

O primeiro algoritmo que veremos (A5.1) é relativamente ineficiente, e sua principal finalidade é preparar o caminho para a discussão de outros métodos mais eficientes.

O algoritmo A5.1 inicia-se com uma etapa de marcação dos nós ativos, por qualquer método apropriado dos vistos no capítulo 3. Na segunda etapa, todos os nós marcados são copiados para o semi-espço novo. Na terceira e última, todos os apontadores de base e das cópias são atualizados, de modo a refletir os deslocamentos havidos.

A atualização dos apontadores baseia-se na idéia do algoritmo de Edwards, de deixar em cada nō copiado o endereço sua cópia. Esta idéia é utilizada por todos os algoritmos deste capítulo.

As áreas INI e LIM, como dantes, contem o endereço e o limite do semi-espço original, e ININOV contem o

procedimento COPIADOR-COM-MARCAÇÃO-PRÉVIA:
efetua

```

1  {marcação dos nós acessíveis}
    MARCA-ACESSÍVEIS:
    {copia para semi-espaco novo}
    áreas E: apontador, T: inteiro de 1 a m;
1  E ← INI; LIVRE ← ININOV;
    repete, enquanto E < LIM,
N  [
    T ← tam(E↑);
    se E↑.marca é 'útil', então
NA [
    move E↑ para LIVRE↑;
    move LIVRE para E↑;
    LIVRE ← LIVRE + T;
N  E ← E + T;
    ]
    {atualiza apontadores}
    procedimento ATUALIZA:
    recebendo área A: campo
    efetua
    [
    se A é apontador, então
    A ← (A↑ como apontador);
    repete, para J desde 1 até CB,
    ATUALIZA(R[J]);
    E ← ININOV;
    repete, enquanto E < LIVRE,
NA [
    repete, para J desde 1 até CB,
    ATUALIZA(E↑.corpo[J]);
    E↑.marca ← 'inútil';
    E ← E + tam(E↑);
    ]
1  INI ↔ ININOV;
    LIM ← LIVRE;
    TROCA-SEMIESPACOS;
  
```

Algoritmo A5.1 - Cópia com marcação prévia. MARCA-ACESSÍVEIS é um procedimento marcador conveniente que torna 'úteis' as marcas dos nós acessíveis. TROCA-SEMIESPACOS deve trocar entre si os significados das operações "↓" e "↑".

endereço do novo semi-espço. Ao fim do algoritmo, LIVRE a ponta para o início da área livre neste semi-espço.

O número de falhas de páginas causadas por este algoritmo é: $P_A + P_B$, no mínimo, durante a marcação; $G_A + G_{AC}$, durante a cópia; e $G_{AC} + P_A + P_B$, na atualização, dando no total $G_A + 2G_{AC} + 2(P_A + P_B)$.

5.3 Cópia com Fila

O primeiro algoritmo de recuperação por cópia a ser publicado já era mais eficiente que A5.1, por combinar numa única etapa as operações de marcação, cópia e atualização.

Esse algoritmo, devido a R.R.Fenichel e J.C.Yochelson {Fen/Yoc 11/69} foi originalmente proposto na forma de um procedimento recursivo. A forma que aqui apresentamos (A5.2) substitui a recursão por uma fila, mas conserva as características essenciais da versão original.

O algoritmo A5.2 é uma adaptação direta do algoritmo de marcação horizontal com fila A3.1. Quando um nó é marcado, seu conteúdo é copiado para o semi-espço novo, e o endereço da cópia é armazenado no original. Todo apontador encontrado é atualizado, após a eventual marcação e cópia do nó que ele designa. A marca-de-nó, neste e nos algoritmos que se seguem, assume os estados 'a copiar' e 'copiado', em vez de 'útil' e 'inútil', respectivamente.

Note-se que o tempo exigido por este algoritmo depende apenas do número de nós, campos e apontadores atingíveis, e não do número total de nós, como A5.1.

O número de falhas de páginas de A5.2 pode ser estimado em $2G_{AC} + P_A + P_B$.

procedimento COPIADOR-COM-FILA-AUXILIAR:

```

efetua
1  [ áreas PE: fila de  $N - \lceil (N-1)/p \rceil$  apontadores,
    inicialmente vazia,
    E: apontador;
    procedimento MOVE-E-ATUALIZA:
    usando X: recebendo área X: campo,
    efetua
    se X é apontador, então
    [ se  $X \uparrow . \text{marca}$  é 'a!copiar', então
      [ move  $X \uparrow$  para LIVRE $\uparrow$ ;  $X \uparrow . \text{marca} \leftarrow$  'copiado';
        move LIVRE para  $X \uparrow . c$ ;
         $X \leftarrow$  LIVRE;
         $PE \leftarrow$  LIVRE;
        LIVRE  $\leftarrow$  LIVRE + tam(LIVRE $\uparrow$ );
      senão
         $X \leftarrow X \uparrow . c$  como apontador;
    LIVRE  $\leftarrow$  ININOV;
    repete, para J desde 1 até  $C_B$ ,
      MOVE-E-ATUALIZA( $R[J]$ );
    repete, enquanto PE não for vazia,
    [ E  $\leftarrow$  PE;
      repete, para J desde 1 até  $E \uparrow . c$ ,
        MOVE-E-ATUALIZA( $E \uparrow . \text{corpo}[J]$ );
    INI  $\leftrightarrow$  ININOV;
    LIM  $\leftarrow$  LIVRE;
    TROCA-SEMI-ESPAÇOS
  ]

```

$C_B + C_A$
 $P_B + P_A$
 N_A
 $P_B + P_A - N_A$
 C_B
 N_A
 C_A

Algoritmo A5.2 - Cópia com fila auxiliar.

Vimos que é possível diminuir o número de falhas do algoritmo A3.1, empilhando os endereços dos filhos de um nó na ocasião em que este é marcado (A3.2). Esta modificação não pode ser aplicada ao algoritmo A5.2, pois quando esses endereços são desempilhados, não há como atualizar os campos de onde eles foram obtidos.

O número de falhas realmente causadas por A5.2 pode ser diminuído, como mencionado no capítulo 3, usando-se uma pilha em substituição à fila PE, ou, melhor ainda, retirando-se dela, de preferência, endereços que designem nós presentemente na memória rápida.

Como a fila PE é relativamente grande, é em geral necessário incluí-la na memória paginada. Isso pode causar um aumento sensível no número de falhas.

5.4 O Algoritmo de Cheney

A principal desvantagem do algoritmo anterior é a necessidade da área auxiliar PE, cujo tamanho, comparado ao do espaço de alocação, está longe de ser desprezível.

Se PE for uma fila, entretanto, os endereços dos nós são retirados dela na mesma sequência em que foram inseridos. Como um endereço é inserido em PE assim que o nó correspondente é copiado para LIVRE \uparrow , essa sequência coincide com a ordem em que as cópias estão dispostas no semi-espaço novo.

Portanto, a fila PE pode ser substituída por um apontador PE que percorra este semi-espaço sequencialmente, com um certo atraso em relação a LIVRE. Os nós entre PE \uparrow e LIVRE \uparrow , a cada momento, serão aqueles cujos endereços es

tariam em PE, no mesmo ponto da execução de A5.2.

Para tanto, basta efetuar as seguintes substituições em A5.2:

PE: fila...	por	PE: endereço de nó, inicialmente ININOV;
$E \Leftarrow PE$	por	$E \leftarrow PE; PE \leftarrow PE + \text{tam}(E \uparrow)$

e eliminar o comando $PE \Leftarrow \text{LIVRE}$. O algoritmo resultante foi proposto por C. J. Cheney em {Che 11/70}.

O tempo de execução é praticamente o mesmo do algoritmo anterior, e o número de falhas é igualmente $2G_{AC+P_A+P_B}$. A eliminação da pilha PE elimina também as falhas eventualmente causadas pelo manuseio da mesma.

5.5 Algoritmos de Cópia Verticais

Os algoritmos verticais de marcação (A3.5, A3.6 e A3.7) podem ser facilmente adaptados para a cópia de estruturas. Essencialmente, é preciso apenas copiar para o semi-espaço novo cada nó que for marcado, deixando em seu lugar o endereço da cópia, e atualizar todo campo apontador encontrado.

No caso do algoritmo A3.5, essas modificações são bastante simples. A cópia é feita imediatamente antes da marcação; a atualização de um apontador pode ser feita quando o mesmo for examinado, se for secundário, ou após desempenhar o nó que o contém, se for primário (algoritmo A5.3). A pilha PE passa a conter os endereços das cópias dos nós parcial-

procedimento COPIADOR-VERTICAL-COM-PILHA:
efetua

1	áreas PE: pilha de N-1 endereços de nós, inicialmente vazia, PI: pilha de N-1 inteiros de 1 a \hat{c} , inicialmente vazia;
	repete, para J desde 1 até C_B ,
C_B	se $R[J]$ é apontador, então
P_B	se $R[J] \downarrow$.marca é 'a copiar', então
	áreas I: inteiro de 1 a \hat{c} ,
	E, A, EN, AN: apontadores;
N_{A1}	$E \leftarrow R[J]$;
	MALHA: repete indefinidamente
N_A	move $E \downarrow$ para LIVRE \uparrow ;
	$EN \leftarrow LIVRE$; $LIVRE \leftarrow LIVRE + \text{tam}(LIVRE \uparrow)$;
	$E \downarrow$.marca \leftarrow 'copiado';
	move EN para $E \downarrow$.corpo;
	$I \leftarrow 1$;
	PROCURA-PRÓXIMO: repete indefinidamente
C_A	se $EN \uparrow$.corpo[I] é apontador, então
P_A	$A \leftarrow EN \uparrow$.corpo[I];
N_{A+}	se $A \downarrow$.marca é 'a copiar', então
	termina PROCURA-PRÓXIMO
$P_A - N_{A+}$	senão
	$EN \uparrow$.corpo[I] $\leftarrow A \downarrow$.corpo como apontador;
C_A	repete,
N_A	enquanto $I = EN \uparrow$.c,
N_{A1}	se PI for vazia, então
	termina MALHA;
	DESEMPILHA:
N_{A+}	$I \leftarrow PI$;
	$AN \leftarrow EN$; $EN \leftarrow PE$;
	$EN \uparrow$.corpo[I] $\leftarrow AN$
$C_A - N_A$	$I \leftarrow I + 1$
	EMPILHA:
N_{A+}	$PE \leftarrow EN$;
	$PI \leftarrow I$;
	PROSSEGUE:
	$E \leftarrow A$
P_B	$R[J] \leftarrow R[J] \downarrow$.corpo como apontador;
	$INI \leftarrow ININOV$; $LIM \leftarrow LIVRE$; TROCA-SEMIESPACOS

Algoritmo A5.3 - Copiador vertical com pilhas explícitas.

mente examinados.

Este copiador vertical é inferior ao de Cheney, notadamente no número de falhas de páginas. As falhas ocorrem na consulta das marcas dos nós $(P_A + P_B)$, na criação das cópias (G_{AC}) , e quando nós são desempilhados. Este último termo é no máximo N_{A+} e costuma ser bem menor, devido à disciplina de pilha.

Outro fator que tende a diminuir este último termo é o fato que, no espaço novo, a probabilidade de um nó estar na mesma página que seus filhos é bastante alta, devido à ordem em que os nós são copiados. Esta característica dos copiadores verticais é extremamente desejável, pois tende a diminuir o número de falhas geradas pelo programa do usuário ao manipular as estruturas.

Assim, embora a recuperação com algoritmos verticais provoque mais transferências de páginas que com o de Cheney, no cômputo geral este pode ser desvantajoso.

5.6 O Algoritmo de Reingold

O algoritmo de Deutsch-Schorr-Waite (A3.6) pode ser adaptado de várias maneiras. Se a pilha PI for conservada, as alterações são semelhantes às descritas na seção anterior.

Como no algoritmo A3.6, a pilha de endereços é substituída por uma cadeia de cópias de nós parcialmente examinadas, obtida invertendo-se o último campo analisado de cada uma (fig. F5.1-b).

Neste caso, o número de acessos será o mesmo do algoritmo com pilha explícita, isto é, menos de $P_A + P_B + G_{AC} + N_{A+}$.

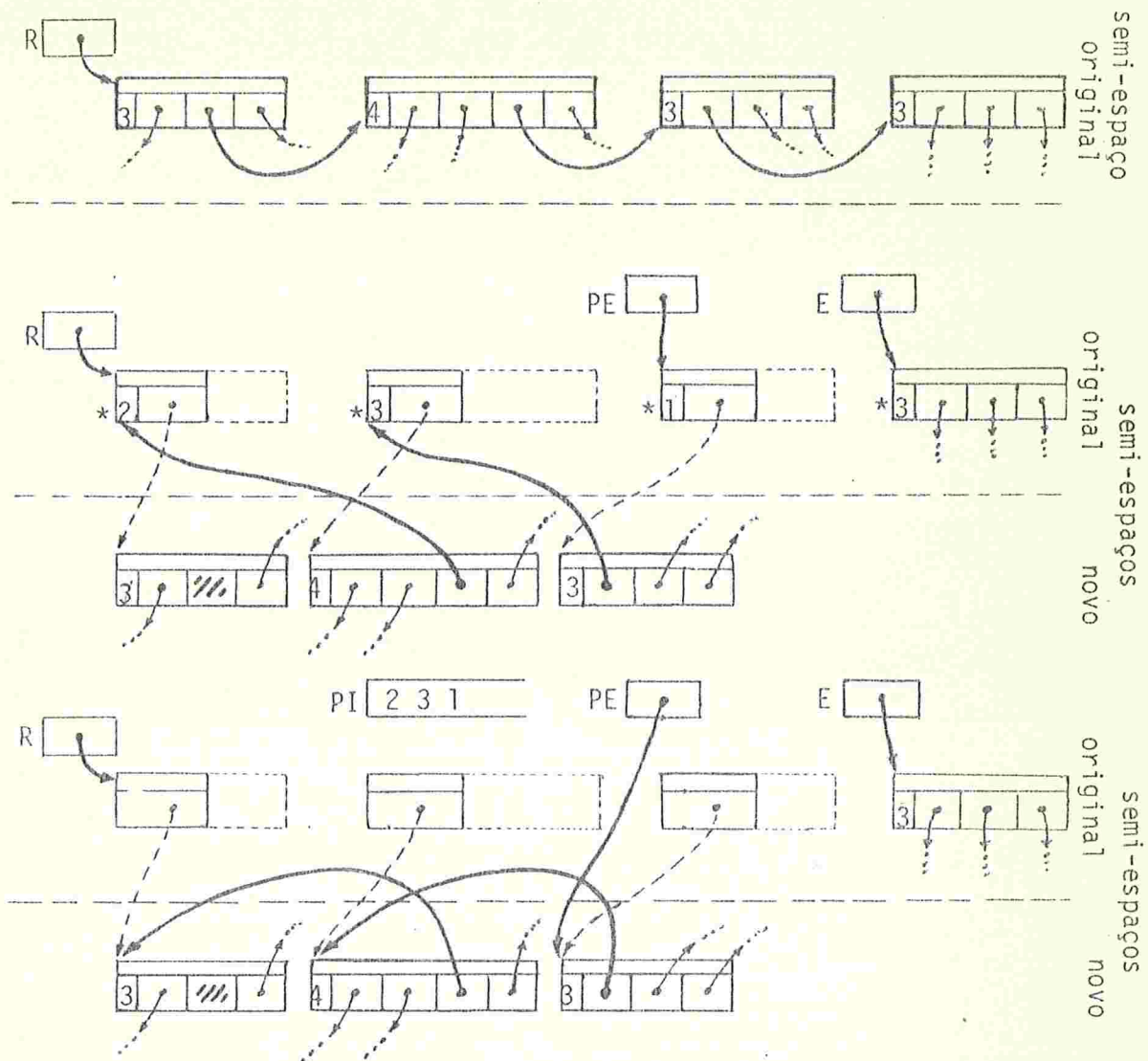


Figura F5.1 - Duas adaptações do algoritmo de Deutch-Schorr-Waite ao problema de copiar uma estrutura. (a) uma cadeia na estrutura original, (b) situação imediatamente anterior à cópia do nó **E**, na versão com pilha **PI** separada; (c) situação no mesmo instante, na versão sem pilha **PI**. Os apontadores em negrito são os "invertidos" pelo algoritmo.

A pilha PI pode ser eliminada, se cada nó E_i tiver espaço suficiente para conter, além do endereço da cópia, um inteiro de 1 a $E_i.c$ (o próprio item $E_i.c$ pode ser usado para esse fim). Podemos neste caso armazenar na área original a entrada de PI correspondente a cada nó. Para que essa entrada possa ser localizada, os apontadores invertidos em cada cópia devem apontar para o semi-espço antigo (fig. F5.1-c).

O preço pago pela eliminação da pilha PI desta forma é um aumento no número de falhas, pois, cada vez que um nó é desempilhado, a área original do mesmo deve também ser consultada. Portanto, o número de falhas pode chegar a $P_A + P_B + G_{AC} + 2N_{A+}$.

Se $\hat{p} \leq 2$, a pilha PI pode ser eliminada sem gerar falhas adicionais, bastando usar o item marca de cada cópia na "pilha", para indicar qual de seus apontadores está invertido. Deve-se tomar cuidado apenas de apagar essa marca ao retirar um elemento da pilha, para que a próxima recuperação possa funcionar corretamente. Esta versão foi apresentada originalmente por E.M.Reingold em {Rei 3/73}.

5.7 O Copiador de Clark

Analisando-se o copiador vertical A5.3, pode-se verificar que todo nó do semi-espço novo que for empilhado terá, ao fim do algoritmo, um filho imediatamente adjacente a ele. De fato, quando um endereço de nó é empilhado pela primeira vez, é porque o nó acabou de ser copiado, e um de seus campos designa um nó ainda por copiar; este último será movido logo em seguida, e portanto, os dois serão adjacentes.

Em {Cla 6/76}, D.W.Clark observou que tais campos podem ser utilizados pelo algoritmo copiador para ligar entre si os nós parcialmente examinados, substituindo assim a pilha PE (fig. 5.2). Para tanto, é necessário lembrar, para cada nó empilhado, qual de seus campos está nessas condições. Isto pode ser conseguido por meio de uma pilha de inteiros, PK.

A pilha PI ainda é necessária, pois o último campo examinado de cada nó da pilha PE nem sempre é o que está invertido, ao contrário do que ocorre no algoritmo de Deutsch-Schorr-Waite.

Apesar de este último dispensar a pilha PK, a técnica de Clark é mais vantajosa, pois permite utilizar a otimização sugerida por ele, e já usada no algoritmo marcador A3.7: não empilhar nós que não tenham mais apontadores a examinar. O algoritmo copiador de Clark (A5.4), graças a esta característica, consegue diminuir o número de falhas de páginas a $P_A + P_B + G_{AC} + K_2$, onde K_2 é o parâmetro ($\leq N_{A+}$) definido na secção 3.10.

As pilhas PK e PI podem ser combinadas numa única pilha PKI, cada entrada da qual pe um par de inteiros $|K, I|$. Note-se que $K < I$ quando ambos são empilhados, de modo que a entrada de PKI correspondente a um nó com $c \geq 2$ campos pode ser codificada em $\lceil \log_2 [c(c-1)/2] \rceil = \lceil \log_2 c + \log_2 (c-1) \rceil - 1$ bits (nós com menos de dois campos nunca são empilhados). Se $\hat{c} \leq 2$, conclui-se que as pilhas PI e PK são dispensáveis.

Como observamos na discussão do marcador de Clark (secção 3.10), podemos substituir "campo" por "apontador" (e \hat{c} por \hat{p}) no paragrafo acima, o que dá uma certa economia de espaço em PK e PI, às custas de algum tempo adicional de execução.

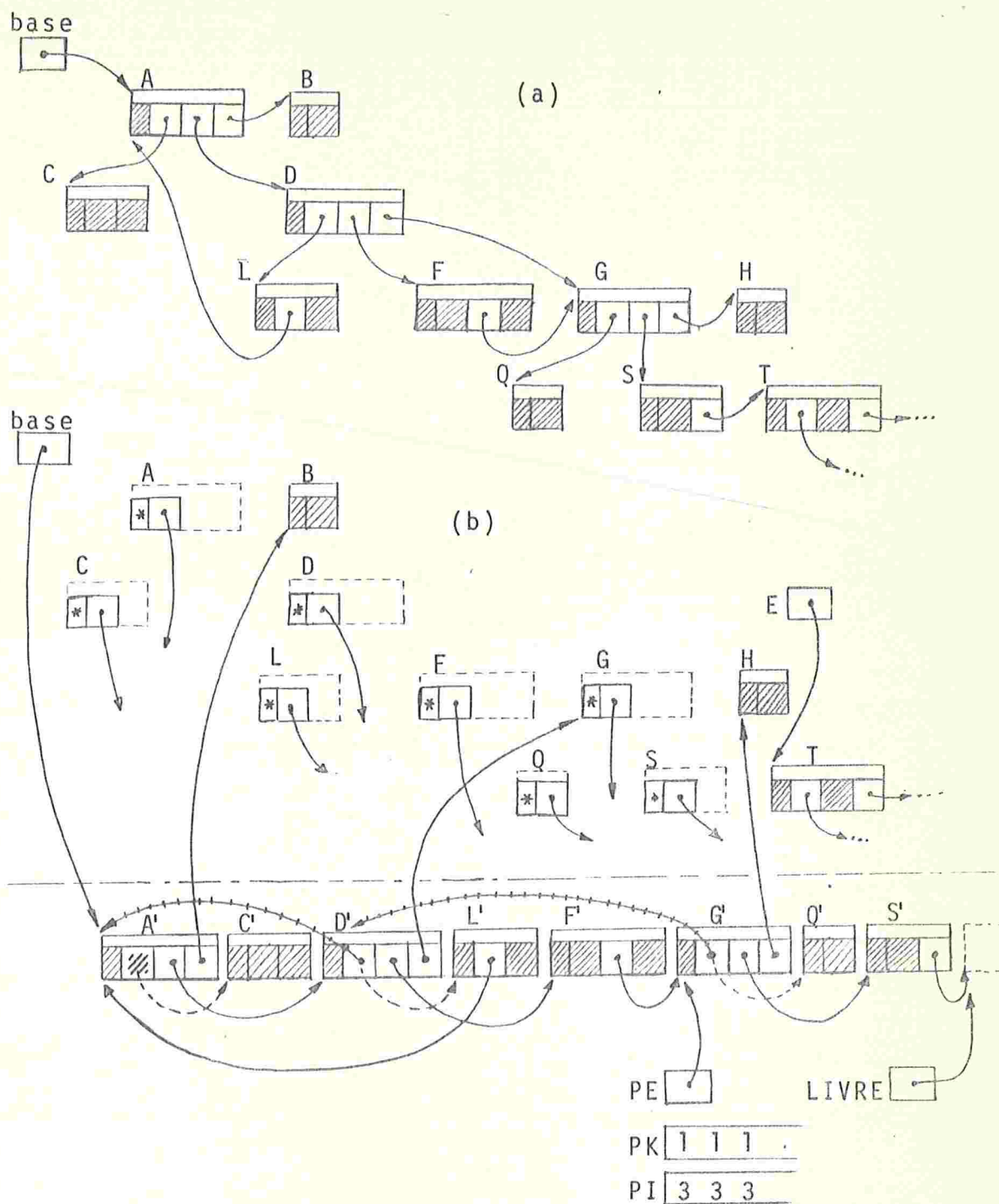


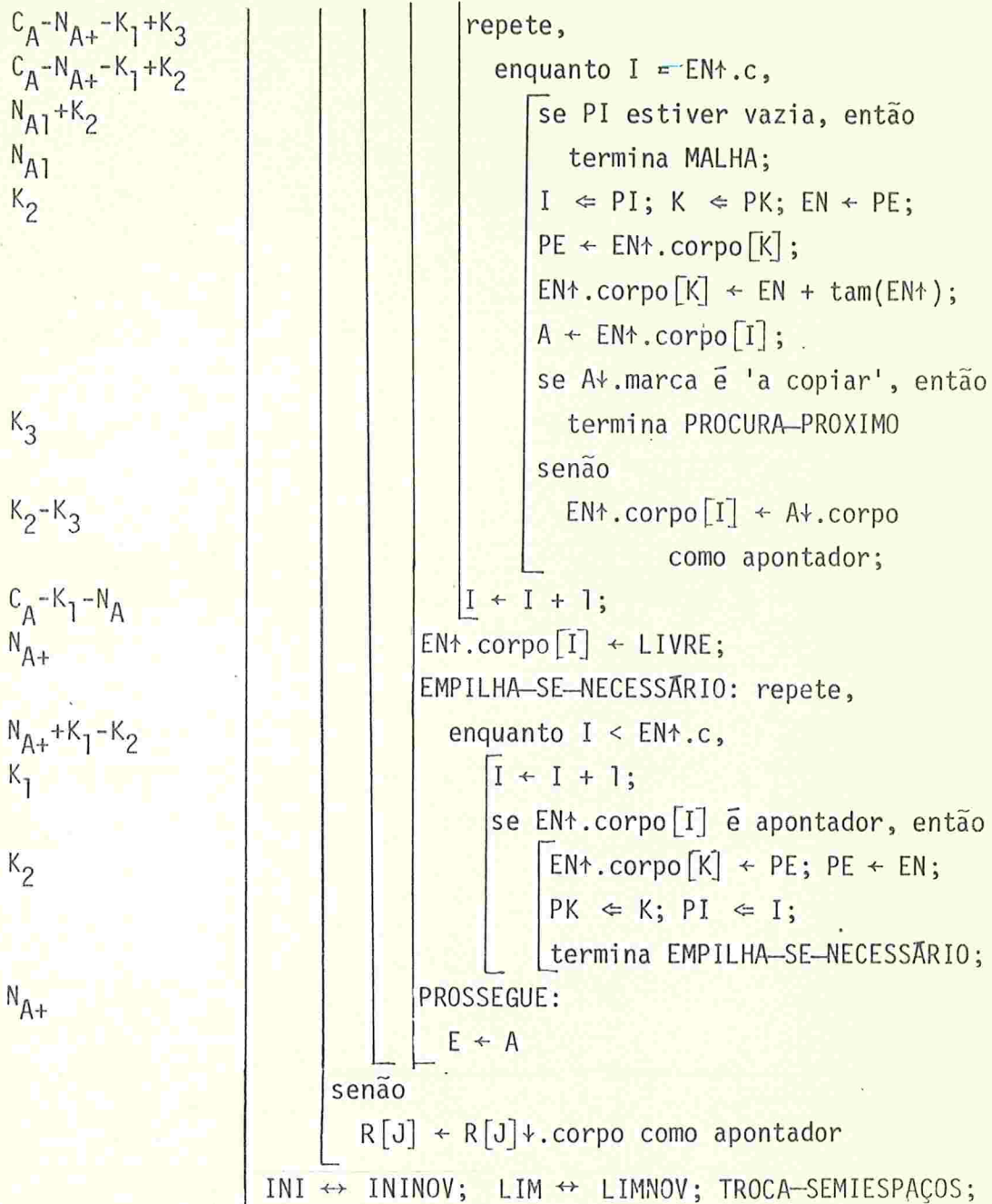
Figura F5.2 - Ilustração do funcionamento do copiador de Clark: (a) situação inicial, (b) situação no início da 9ª iteração de MALHA. As setas tracejadas são apontadores temporariamente substituídos pelas ligações da pilha (↔↔↔↔).

procedimento COPIADOR-CLARK:

efetua

1	áreas PK: pilha de N-1 inteiros de 1 a $\hat{c}-1$, inicialmente vazia,
	PI: pilha de N-1 inteiros de 2 a \hat{c} , inicialmente vazia,
	PE: união uniforme disjunta de ('nada', apontador) inicialmente 'nada';
	repete, para J desde 1 até C_B ,
C_B	se $R[J]$ é apontador, então
P_B	se $R[J].\text{marca}$ é 'a copiar', então
N_{A1}	áreas E, EN, A: apontadores,
	I, K: inteiros de 1 a \hat{c} ;
	$E \leftarrow R[J]; R[J] \leftarrow \text{LIVRE};$
	MALHA: repete indefinidamente
N_A	move $E \uparrow$ para $\text{LIVRE} \uparrow$;
	$EN \leftarrow \text{LIVRE}; \text{LIVRE} \leftarrow \text{LIVRE} + \text{tam}(\text{LIVRE} \uparrow);$
	$E \uparrow.\text{marca} \leftarrow \text{'copiado'}; \text{move EN para } E \uparrow.\text{corpo};$
	$I \leftarrow 1; K \leftarrow \hat{c};$
	PROCURA-PRÓXIMO: repete indefinidamente
$C_A - K_1$	se $EN \uparrow.\text{corpo}[I]$ é apontador, então
$P_A - K_2$	$A \leftarrow EN \uparrow.\text{corpo}[I];$
	se $A \uparrow.\text{marca}$ é 'a copiar', então
$N_{A+} - K_3$	se $K > I$, então
K_4	$K \leftarrow I;$
$N_{A+} - K_3$	termina PROCURA-PRÓXIMO;
	senão
$P_A - N_{A+} - K_2 + K_3$	$EN \uparrow.\text{corpo}[I] \leftarrow A \uparrow.\text{corpo}$ como aponta-apontador;

Algoritmo A5.4 - O copiador de Clark. Os parâmetros K_1 , K_2 e K_3 foram definidos na seção 3.10. K_4 é o número de nós da árvore de percurso que tem pelo menos um filho. (continua)



Algoritmo A5.4 (conclusão)

5.8 Nós e Campos Amorfos; Nós Sobrepostos

Até agora, tratamos apenas de estruturas com nós auto-descritos, disjuntos dois a dois, e com espaço suficiente para armazenar um endereço.

Essas restrições são muito importantes. A primeira pode ser eliminada, se os campos forem auto-descritos e as duas outras restrições permanecerem. Neste caso, o algoritmo copiador com marcação prévia (A5.1) exige uma marca em cada campo, e que este tenha tamanho suficiente para conter um endereço; o percurso do semi-espço antigo, e a cópia, são feitos campo-a-campo.

O algoritmo com fila auxiliar (A5.2) e os copiadores verticais exigem apenas uma marca em cada nó, e uma fila ou pilha adicional de inteiros para armazenar o número de campos de cada nó empilhado (no algoritmo de Reingold, esta informação pode ser armazenada nos apontadores invertidos). O algoritmo de Cheney dispensa essa pilha, e percorre campo-a-campo o semi-espço novo.

Se os campos também forem amorfos, o algoritmo com marcação prévia e o de Cheney não podem ser usados. Os demais continuam aplicáveis, guardando-se na fila ou pilha adicional o descritor completo de cada nó empilhado (em vez do número de campos).

Porém, nós amorfos são usados quando os mesmos podem ser sobrepostos. Nesse caso, apenas o copiador com marcação prévia (A3.1) pode ser usado. Todos os outros falham devido ao problema ilustrado na figura F5.3: se o nó B for movido antes do nó C, este último não poderá ser copiado.

Para evitar esta situação, é necessário percorrer

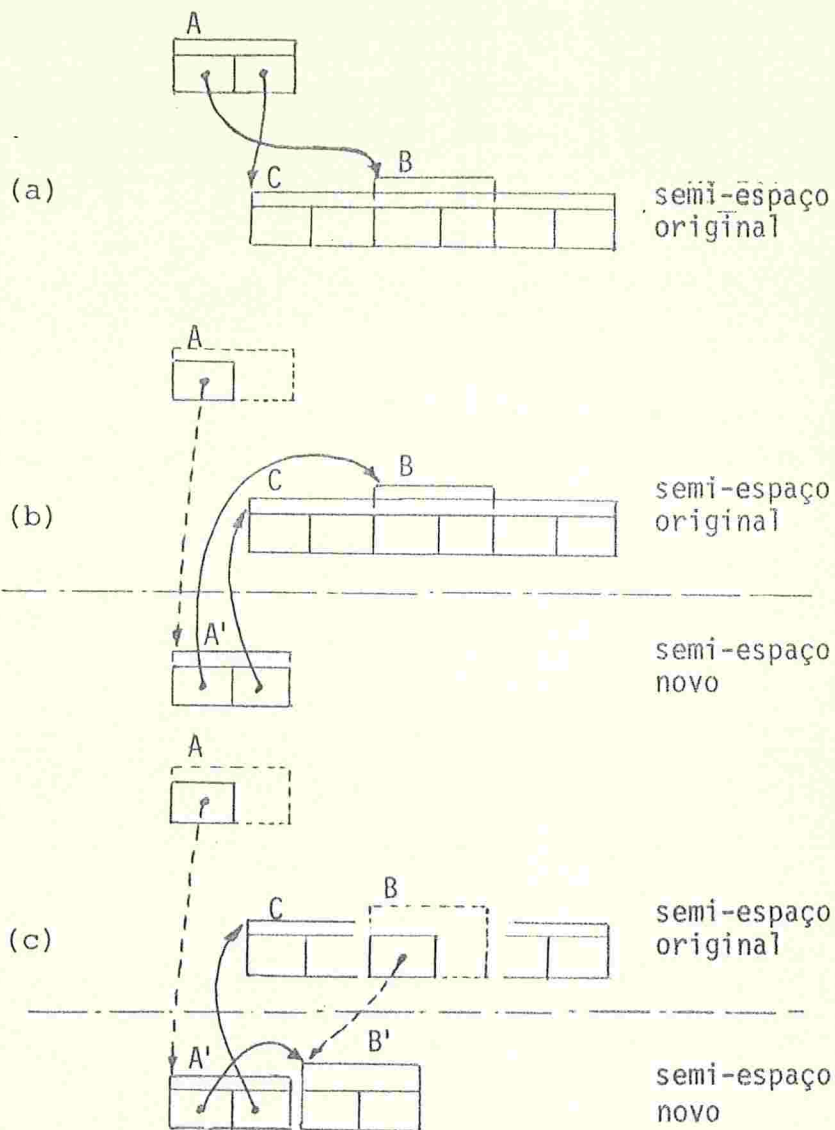


Figura F5.3 - O problema apresentado por nós superpostos nos algoritmos de cópia. Se B for movido antes de C, não será possível copiar este último sem destruir a relação entre B e C.

as estruturas, localizando todos os nós ativos, antes de começar a cópia, e mover cada bloco para o espaço novo, sem alterar a ordem dos nós nele contidos.

Além do algoritmo A5.1, pode-se adaptar a esse fim o compactador de Zave (A4.10), que tem a vantagem de funcionar mesmo com campos amorfos. As modificações são triviais : basta manter LIVRE apontando para o semi-espaço novo, na fase de movimentação dos blocos.

Se a estrutura tiver nós pequenos demais para conter um endereço, apenas o algoritmo de Zave, adaptado como descrito acima, continua aplicável; os demais falham por não poderem deixar no registro original o endereço da cópia.

Pode-se pensar, neste caso, em efetuar a cópia por segmentos, tal como foi discutido nos capítulos de marcação e compactação; mesmo assim, apenas o algoritmo com marcação prévia pode ser salvo.

O problema causado por essa técnica nos outros copiadores é que um nó pode estar contido em dois segmentos consecutivos, e estes podem ser copiados, para posições não adjacentes, antes desse nó ser encontrado. Esse problema pode ser resolvido obrigando-se todo segmento a conter apenas nós completos, ou apenas uma parte de um único nó; mas isto raramente é praticável.

Note-se que nós e campos pequenos não apresentam nenhum problema especial ao algoritmo de Zave.

CAPÍTULO VI - RECUPERADORES EM TEMPO REAL

6.1 Introdução

Os algoritmos vistos nos capítulos anteriores exigem que o programa usuário seja interrompido durante toda a fase de recuperação. Em sistemas de grande porte, essa pausa pode durar segundos, ou mesmo minutos. Suas consequências podem ir desde irritação dos usuários de programas interativos, a desastres sérios em aplicações de controle em tempo real.

Nos últimos anos, este problema tem sido bastante estudado, e publicaram-se vários algoritmos recuperadores que podem ser executados simultâneamente com o programa usuário, ou alternando-se a ele durante breves períodos. Com tais algoritmos, a velocidade aparente do sistema é ligeiramente reduzida, mas evitam-se pausas demoradas. Mais do que isso: é possível garantir que o intervalo de tempo entre duas operações consecutivas executadas pelo usuário será limitado por uma constante "razoável", que independe do tamanho do espaço de alocação e do das estruturas ativas.

Denominaremos tais algoritmos de recuperadores em tempo real. Embora muitos deles tenham sido planejados para serem executados num processador próprio, paralelamente ao programa usuário e raramente interferindo com ele, todos podem ser usados em sistemas de processador único, recorrendo-se a técnicas conhecidas de multiprogramação e tempo com-partilhado.

Os algoritmos vistos até aqui, para serem transfor

mados em recuperadores de tempo real, precisam sofrer diversas modificações. As estruturas podem ser modificadas pelo programa usuário no decorrer da recuperação, e é preciso garantir que esta não tenha que ser re-iniciada cada vez que isso acontece. As operações básicas do usuário (criação de novos nós, alteração e consulta de campos) podem encontrar nós e apontadores relocados, ou temporariamente modificados, e é preciso estendê-las para cobrir essas situações. Finalmente, é preciso sincronizar a execução do recuperador com a do programa usuário, para que os dois não tentem realizar simultaneamente operações incompatíveis sobre os mesmos objetos.

A idéia da recuperação em tempo real é atribuída a M.Minski por Knuth ({Knu 68 - 2.3.5 ex. 12}). O interesse generalizado na mesma parece ter sido despertado em 1975, quando Guy L.Steele Jr., então estudante de graduação em Harvard, publicou um recuperador e compactador em tempo real {Ste 9/75}.

Muitos foram os algoritmos que se seguiram, e este assunto ainda está sendo pesquisado, de modo que avanços significativos podem ser previstos no futuro próximo.

6.2 Sincronização de Processos

Neste capítulo, o recuperador e o programa usuário serão considerados dois processos, isto é, dois "algoritmos" cuja execução nem sempre termina.

Em princípio, os dois são executados em paralelo: isto é, as operações efetuadas por um são simultâneas, ou arbitrariamente intercaladas, com as do outro. Não faremos a priori nenhuma suposição quanto à velocidade com que cada processo caminha; admitiremos apenas que, a cada instante, pelo me

nos um deles está sendo executado.

As áreas de memória locais a um processo são as manipuladas exclusivamente por ele. Áreas que podem vir a ser consultadas ou alteradas por mais de um processo são ditas globais aos mesmos. No nosso caso, terão forçosamente que ser globais os espaços de base e de alocação.

Se dois processos tentam alterar ou consultar simultaneamente uma mesma área global, o resultado da operação dependerá do sistema em questão, e, em muitos casos, de fatores totalmente aleatórios. É indispensável, portanto, que tenhamos algum meio de impedir tais acessos simultâneos.

O meio que adotaremos para tal fim, nos algoritmos que se seguem, é a inclusão nos mesmos de operações explícitas de sincronização, com que um processo pode interromper o outro quando houver possibilidade de conflito. Estas operações são baseadas no conceito de semáforo, proposto por E.W. Dijkstra em {Dij 11/68}.

Para nós, um semáforo α será basicamente uma área, global aos processos que devem ser sincronizados, que pode conter apenas os dois valores 'livre' e 'ocupado', e que pode ser manipulada apenas pelas operações $\text{reserva } \alpha$ e $\text{libera } \alpha$.

A operação $\text{reserva } \alpha$ pode ser executada apenas se o conteúdo do semáforo for 'livre', e seu efeito é torná-lo 'ocupado'. Se o semáforo já estiver 'ocupado', o processo que tentou executar $\text{reserva } \alpha$ é suspenso. Eventualmente, após um período não especificado, o mesmo é retomado, a partir da operação $\text{reserva } \alpha$; se α ainda estiver ocupado o processo é novamente suspenso, e tudo se repete mais uma vez.

A operação $\text{libera } \alpha$ deixa o semáforo α 'livre',

independentemente de seu estado no momento. Assim, um processo que foi suspenso por executar $\text{reserva } \alpha$ com α 'ocupado' só poderá prosseguir se um outro processo executar $\text{libera } \alpha$.

Essas duas operações devem ser mutuamente exclusivas, isto é, o sistema deve suspender automaticamente um processo que tente executar $\text{reserva } \alpha$ ou $\text{libera } \alpha$, enquanto outro estiver efetuando qualquer dessas operações.

O conteúdo inicial de um semáforo será sempre especificado na declaração do mesmo.

A título de exemplo, vamos mostrar como essas duas operações podem ser usadas para evitar que dois processos tentem manipular ao mesmo tempo uma área global. Para tanto, declara-se também uma área α :semáforo inicialmente 'livre'. Em cada processo, toda referência à área global deve ser precedida por uma instrução $\text{reserva } \alpha$, e seguida por $\text{libera } \alpha$.

Desta forma, se dois processos tentarem manipular ao mesmo tempo a área global, o primeiro a executar $\text{reserva } \alpha$ encontrará o semáforo α 'livre', e poderá prosseguir; enquanto que os outros, encontrando α 'ocupado', serão suspensos, até que o primeiro termine suas manipulações da área global e efetue a operação $\text{libera } \alpha$.

Se α é um semáforo como o acima descrito, uma sequência de instruções quaisquer que não afetem α , precedida por $\text{reserva } \alpha$, e seguida por $\text{libera } \alpha$, é denominada região α -crítica, (ou simplesmente região crítica se α for subentendido).

Duas regiões α -críticas, em processos distintos, são mutuamente exclusivas, isto é, enquanto uma delas estiver sendo executada, a outra não poderá ser iniciada. No exemplo que demos acima, cada operação envolvendo a variável em questão

foi transformada numa região α -crítica.

Comandos condicionais executados por um processo também podem necessitar do uso de semáforos, se o teste envolver áreas globais, para evitar que o outro processo altere estas (e portanto invalide o teste), antes do comando ser completado. Regiões críticas, neste e em outros casos, são o meio mais simples de evitar conflitos entre processos.

Além do comando reserva, outras condições podem determinar a suspensão de um processo. Se o número de processadores for menor que o número de processos, pode ser necessário suspender alguns destes para que os outros possam continuar.

Os critérios para a escolha do próximo processo a ser suspenso ou ativado, e do momento em que isso deve ocorrer, serão chamados de critérios de prioridade. Esses critérios devem ser escolhidos de modo a assegurar que todos os processos avancem de maneira "satisfatória".

6.3 Sincronização Eficiente entre Recuperador e Usuário

Se usarmos regiões críticas para sincronizar os processos recuperador e usuário, este último poderá ser suspenso toda vez que ocorrer um conflito. Com os critérios de prioridade mais favoráveis, a duração máxima de uma dessas suspensões será igual ao tempo exigido pelo recuperador para executar a maior de suas regiões críticas. Se quisermos evitar suspensões demoradas, então, este tempo deve ser mantido pequeno.

O processo usuário pode ser suspenso também quando o alocador encontra esgotado o espaço livre. Neste caso, o pro

cesso usuário é forçado a esperar que o recuperador colete áreas inativas suficientes para a alocação. Para eliminar pausas deste tipo, é necessário que o recuperador avance a uma velocidade suficiente para repor o espaço livre à mesma razão que o alocador o consome. Isto exige, além de critérios prioridade adequados, que o programa usuário não crie nós em demasia dentro de suas seções críticas.

Nos algoritmos que apresentaremos neste capítulo, usaremos em geral um semáforo único para controlar o acesso aos espaços de base e de alocação. As regiões críticas do processo usuário serão limitadas às operações que afetam esses dois espaços, e as do recuperador terão em geral duração comparável a essas.

O uso de um único semáforo controlando todo o espaço de alocação é ineficiente, em sistemas com dois ou mais processadores, pois a probabilidade de ocorrer um conflito é elevada, e nessas ocasiões apenas um dos processadores pode ser usado. A execução em paralelo só é possível enquanto o programa usuário estiver manipulando áreas situadas fora desses dois espaços.

Na verdade, é possível diminuir bastante esses conflitos, usando-se técnicas mais sofisticadas de sincronização. Entretanto, é impossível usar corretamente essas técnicas, que permitem um grau elevado de paralelismo entre os dois processos, com base apenas na intuição. A única maneira de justificar o funcionamento de tais algoritmos é por meio de longas e tediosas demonstrações formais, que aliás, como pode ser constatado na bibliografia ({Gri 12/77}), não são infalíveis.

Portanto, procuraremos simplificar ao máximo os métodos de sincronização usados nos algoritmos, e discutiremos no

final do capítulo as idéias básicas dessas técnicas mais sofisticadas.

6.4 O Algoritmo de Steele

O primeiro algoritmo que estudaremos é uma variante do proposto por Guy Steele Jr. em {Ste 9/75}. O método usado é basicamente a marcação horizontal com pilha (A3.1), seguida de compactação e coleta.

A compactação é feita usando-se a técnica de Edwards (A4.7), e a coleta usa lista livre (A4.1). Essas duas características restringem o algoritmo de Steele a situações com nós de tamanho uniforme.

Para sua execução simultânea, tanto os algoritmos acima quanto o processo usuário devem disciplinar seus acessos aos espaços de base e de alocação. Usaremos para esse fim um semáforo S ; toda instrução que puder eventualmente alterar ou consultar um desses espaços deve ser englobada por uma região S -crítica.

Duas sequências de instruções, uma em cada processo, cuja intercalação pode produzir resultados incorretos, também devem ser transformadas em duas regiões S -críticas. Como estas não podem ser muito extensas, entretanto, teremos que recorrer a outras técnicas para evitar os problemas que podem ocorrer dessas intercalações.

Um desses problemas surge quando o usuário tenta manipular um nó $A+$, que foi relocado pelo recuperador, antes deste atualizar o apontador A . O usuário deve poder detectar esta situação, e, caso ela ocorra, deve atualizar A por conta própria, antes de manipular $A+$.

Um segundo problema ocorre quando o usuário efetua uma atribuição entre dois campos, de base ou de nó, durante a etapa de marcação. Suponha, por exemplo, que o recuperador já marcou toda a estrutura apontada pelo campo de base A, e que, antes que ele possa examinar o campo B, o usuário efetue $A \leftarrow B$; $B \leftarrow \text{'nada'}$. Como o recuperador não voltará a examinar A antes da próxima coleta, o nó $A \downarrow$ pode continuar sem marca, e ser recolhido à lista livre apesar de ser atingível por A.

Um problema similar ocorre durante a etapa de atualização dos apontadores. Se o recuperador atualizou A, mas não B, e o usuário efetua $A \leftarrow B$, A pode continuar desatualizado até a coleta seguinte, quando o nó $A \downarrow$ poderá ser recolhido à lista livre.

Para resolver esses dois problemas, após a atribuição $A \leftarrow B$ o usuário deve atualizar A, se $A \downarrow$ tiver sido relocado, e empilhá-lo, marcando $A \downarrow$, se este não estiver marcado.

Os nós criados pelo usuário no decorrer da etapa de compactação podem não ser relocados, mesmo que haja uma área livre de endereço menor; portanto, o espaço livre geralmente é formado de várias áreas não consecutivas, tornando obrigatório o uso de uma lista livre.

No início da coleta, a lista livre geralmente não estará vazia; e é necessário que ela continue assim, pois o usuário pode criar novos nós durante a coleta. Os nós que ainda estão na lista livre não podem ser recolhidos como os inativos, pois isso destruiria a estrutura daquela. Portanto, o recuperador deve ser capaz de distinguir esses nós no percurso sequencial da memória.

Na descrição do recuperador de Steele (A6.1), suporemos que cada nó possui um item marca, que pode assumir quatro valores, relacionados a seguir com seus significados aproximados:

'útil' : Indica que o nó é acessível a partir de um campo de base, ou que o foi em algum instante recente.

'inútil' : Ao terminar a marcação, indica que o nó é seguramente inacessível ao usuário. Durante a coleta, pode significar que o nó era útil e já foi examinado por essa etapa.

'livre' : Indica um nó da lista livre. O primeiro campo aponta para o nó seguinte da mesma.

'relocado' : Indica que o conteúdo original deste nó foi copiado para outra área, cujo endereço está no primeiro campo deste nó.

Os estados 'livre' e 'relocado' poderiam ser indicados por dois átomos especiais, reservados para este fim, armazenados no segundo campo de cada nó. Desta forma, marca poderia ser reduzida a um único bit.

Para efetuar uma atribuição $R[I] \leftarrow R[J]$ entre dois campos de base, o usuário deve recorrer ao procedimento **ATRIBUI** (A6.2). Para efetuar atribuições de e para campos de

modos marca-de-não: união rotulada uniforme de ('útil', 'inútil',
'livre', 'relocado');

não: [marca: marca-de-não,
corpo: sequência de \hat{c} campos];

áreas PE: pilha de N endereços de não, inicialmente vazia,

S: semáforo, inicialmente 'livre',

ECOL: endereço de não, inicialmente INI;

procedimento EXAMINA {a ser invocado apenas em regiões S-críticas}:

recebendo área A: campo {de base ou de não},

efetua

$$\begin{array}{l} C_B + \hat{C}N_E^i \\ N_E^i \end{array} \left[\begin{array}{l} \text{se A não é átomo,} \\ \text{e } A > \text{ECOL,} \\ \text{e } A \downarrow .\text{marca} \leftarrow \text{'inútil'}, \text{ então} \\ \quad \left[\begin{array}{l} A \downarrow .\text{marca} \leftarrow \text{'útil'}; \\ \text{PE} \leftarrow A; \end{array} \right. \end{array} \right.$$

procedimento RECUPERADOR-STEELE:

efetua

$$1 \left[\begin{array}{l} \text{repete indefinidamente} \\ \quad \left[\begin{array}{l} \text{MARCA-NÓS-ATIVOS;} \\ \text{RELOCA;} \\ \text{ATUALIZA-APONTADORES;} \\ \text{RECOLHE} \end{array} \right. \end{array} \right.$$

Algoritmo A6.1 - O recuperador de Steele. LIM contém o limite do espaço de alocação. As estatísticas incluem apenas as operações efetuadas pelo processo recuperador, em cada ciclo; os parâmetros usados nas mesmas são explicados no texto. (continua)

procedimento MARCA-NÓS-ATIVOS:

efetua

```

[ área E: apontador;
1  {M0}
  reserva S; ECOL ← LIM; libera S;
  repete, para J desde 1 até CB,
CB  [ {M1}
      reserva S;
      EXAMINA(R[J]);
      libera S;
MALHA: repete indefinidamente
NEi+1 [ {M2}
        reserva S;
        se PE for vazia, então
1      [ libera S;
          termina MALHA;
NEi  E ← PE;
        libera S;
        repete, para J desde 1 até  $\hat{C}$ ,
CNEi  [ {M3}
          reserva S;
          EXAMINA(E+.corpo[J]);
          libera S;
[M4]

```

Algoritmo A6.1 (continuação)

procedimento RELOCA:

efetua

```

1  áreas E, EL: apontadores,
    MC: marca-de-não;
    E ← LIM; EL ← INI;
    MALHA: repete indefinidamente
NADi+1  PROCURA-NÃO-INÚTIL: repete indefinidamente
K1i      {R1}
          reserva S; MC ← EL↑.marca; libera S;
          se MC = 'inútil', então
NADi      termina PROCURA-NÃO-INÚTIL;
          EL ← EL + tam(não);
          se EL ≥ E, então
          termina MALHA;
N-K1i    PROCURA-NÃO-ÚTIL: repete indefinidamente
          E ← E - tam(não);
          se E ≤ EL, então
          termina MALHA;
          {R2}
          reserva S; MC ← E↑.marca; libera S;
          se MC = 'útil', então
          termina PROCURA-NÃO-ÚTIL;
NADi      {R3}
          reserva S;
          move E↑ para EL↑;
          E↑.marca ← 'relocado';
          move EL para E↑.corpo;
          libera S;
          EL ← EL + tam(não)

```

Algoritmo A6.1 (continuação). N_{AD}^i é o número de nós relocados no ciclo i .

procedimento ATUALIZA {a ser invocado apenas em regiões S-críticas}:

recebendo área A: campo {de base ou de nō},

efetua

$\langle \bar{N}_A$
 $P_{BJ}^i + P_{AJ}^i$

[se A é apontador e A↑.marca = 'relocado', então A ← A↑.corpo como apontador;
---	--

procedimento ATUALIZA-APONTADORES:

efetua

1
 C_B

[repete, para J desde 1 até C_B , <table border="0"> <tr> <td style="vertical-align: middle; padding-right: 10px;">[</td> <td> $\{A_1\}$ reserva S; ATUALIZA(R[J]); libera S; </td> </tr> </table>	[$\{A_1\}$ reserva S; ATUALIZA(R[J]); libera S;
[$\{A_1\}$ reserva S; ATUALIZA(R[J]); libera S;		

áreas E: endereço de nō, MC: marca-de-nō;
 $E \leftarrow \text{INI};$

1
 N

[repete, enquanto $E < \text{LIM}$, <table border="0"> <tr> <td style="vertical-align: middle; padding-right: 10px;">[</td> <td> $\{A_2\}$ reserva S; MC ← E↑.marca; libera S; se MC = 'útil', então <table border="0"> <tr> <td style="vertical-align: middle; padding-right: 10px;">[</td> <td> repete, para J desde 1 até \hat{C}, <table border="0"> <tr> <td style="vertical-align: middle; padding-right: 10px;">[</td> <td> $\{A_3\}$ reserva S; ATUALIZA(E↑.corpo[J]); libera S; </td> </tr> </table> </td> </tr> </table> </td> </tr> </table>	[$\{A_2\}$ reserva S; MC ← E↑.marca; libera S; se MC = 'útil', então <table border="0"> <tr> <td style="vertical-align: middle; padding-right: 10px;">[</td> <td> repete, para J desde 1 até \hat{C}, <table border="0"> <tr> <td style="vertical-align: middle; padding-right: 10px;">[</td> <td> $\{A_3\}$ reserva S; ATUALIZA(E↑.corpo[J]); libera S; </td> </tr> </table> </td> </tr> </table>	[repete, para J desde 1 até \hat{C} , <table border="0"> <tr> <td style="vertical-align: middle; padding-right: 10px;">[</td> <td> $\{A_3\}$ reserva S; ATUALIZA(E↑.corpo[J]); libera S; </td> </tr> </table>	[$\{A_3\}$ reserva S; ATUALIZA(E↑.corpo[J]); libera S;
[$\{A_2\}$ reserva S; MC ← E↑.marca; libera S; se MC = 'útil', então <table border="0"> <tr> <td style="vertical-align: middle; padding-right: 10px;">[</td> <td> repete, para J desde 1 até \hat{C}, <table border="0"> <tr> <td style="vertical-align: middle; padding-right: 10px;">[</td> <td> $\{A_3\}$ reserva S; ATUALIZA(E↑.corpo[J]); libera S; </td> </tr> </table> </td> </tr> </table>	[repete, para J desde 1 até \hat{C} , <table border="0"> <tr> <td style="vertical-align: middle; padding-right: 10px;">[</td> <td> $\{A_3\}$ reserva S; ATUALIZA(E↑.corpo[J]); libera S; </td> </tr> </table>	[$\{A_3\}$ reserva S; ATUALIZA(E↑.corpo[J]); libera S;		
[repete, para J desde 1 até \hat{C} , <table border="0"> <tr> <td style="vertical-align: middle; padding-right: 10px;">[</td> <td> $\{A_3\}$ reserva S; ATUALIZA(E↑.corpo[J]); libera S; </td> </tr> </table>	[$\{A_3\}$ reserva S; ATUALIZA(E↑.corpo[J]); libera S;				
[$\{A_3\}$ reserva S; ATUALIZA(E↑.corpo[J]); libera S;						

$E \leftarrow E + \text{tam}(\text{nō})$

$\langle \bar{N}_A$
 $\langle \hat{C} \bar{N}_A$

Algoritmo A6.1 - (continua). $P_{AJ}^i + P_{BJ}^i$ é o número de apontadores acessíveis e de base, no ciclo i, que designam nōs relocados.

procedimento COLETA:

```

    efetua
1      MALHA: repete indefinidamente
N+1    {C1}
        reserva S;
        se ECOL ≤ INI, então
1          libera S;
          termina MALHA;
N      ECOL ← ECOL - tam(n̄o);
        se ECOL↓.marca ≠ 'livre', então
>N-NLi      se ECOL↓.marca ≠ 'útil', então
<N-NLi-NEi      move LIVRE para ECOL↓.corpo;
                  LIVRE ← ECOL;
                  ECOL↓.marca ← 'livre'
        senão
<NEi-NXi      ECOL↓.marca ← 'inútil';
N      libera S;

```

Algoritmo A6.1 (conclusão)

procedimento ALOCADOR:

```

recebendo área A: campo {de base},
efetua
[ ESPERA: repete indefinidamente
  [ reserva S;
    se LIVRE ≠ 'nada', então termina ESPERA;
    libera S; ESPERA;
  A ← LIVRE;
  LIVRE ← LIVRE↑.corpo como apontador;
  repete, para J desde 1 até c,
    A↑.corpo[J] ← 'nada';
  se A < ECOL, então
    A↑.marca ← 'útil'
  senão
    A↑.marca ← 'inútil';
  libera S;

```

Algoritmo A6.2 - Rotinas que o usuário deve invocar para criar nós, e alterar ou comparar apontadores, compatíveis com o recuperador de Steele. (continua)

procedimento ATRIBUI:

recebendo áreas A,B: campos {de base},
efetua

```
[reserva S;
  A ← B;
  EXAMINA(A); ATUALIZA(A);
  libera S;
```

procedimento EXTRAI-CAMPO:

recebendo J: inteiro de 1 a \hat{c} ,
e áreas A,B: campos {de base},
efetua

```
[reserva S;
  ATUALIZA(B);
  A ← B↑.corpo[J];
  EXAMINA(A); ATUALIZA(A);
  libera S;
```

procedimento ALTERA-CAMPO:

recebendo J: inteiro de 1 a \hat{c} ,
e áreas A,B: campos {de base},
efetua

```
[reserva S;
  ATUALIZA(A);
  A↑.corpo[J] ← B;
  EXAMINA(A↑.corpo[J]); ATUALIZA(A↑.corpo[J]);
  libera S;
```

procedimento COMPARA-CAMPOS:

recebendo áreas A,B: campos {de base},
R: booleano {resultado da comparação},
efetua

```
[reserva S;
  ATUALIZA(A); ATUALIZA(B);
  R ← A = B;
  libera S;
```

Algoritmo A6.2 (continuação)

nós devem ser usados os procedimentos EXTRAI-CAMPO e ALTERA-CAMPO, ou combinações dos mesmos.

Para testar se um campo de base ou de nó é atômico, e para utilizar seu conteúdo, caso o seja, este deve ser atribuído primeiramente a um campo de base usando EXTRAI-CAMPO. O teste, e demais operações, podem ser feitas sobre o campo de base, com a precaução de envolvê-los em seções S-críticas. Esta última precaução é também necessária para atribuir um dado atômico a um campo de base.

Um cuidado especial é necessário ao comparar dois endereços de nós, pois um deles pode apontar para a nova posição de um nó relocado, enquanto que o outro pode designar a antiga. O procedimento COMPARA-CAMPOS deve ser usado para efetuar essa operação. Note-se que a única comparação permitida com endereços é o teste de igualdade, uma vez que a relocação não preserva a ordem relativa dos nós.

Os procedimentos de atribuição precisam marcar e empilhar os nós 'inúteis' encontrados, apenas durante a etapa de marcação. Nas etapas de relocação e atualização, todos os nós ativos são 'úteis' ou 'relocados', de modo que essa preocupação é supérflua. Durante a fase de coleta, entretanto, as marcas são apagadas, e o usuário pode encontrar nós 'inúteis' nessa etapa.

Da mesma forma, o alocador precisa marcar os nós criados só quando os mesmos ainda não tiverem sido examinados pelo coletor.

No algoritmo A6.1, a variável global ECOL indica constantemente o progresso da coleta. Todos os nós entre ECOL (inclusive) e LIM (exclusive) foram coletados no ciclo corrente, e os demais ainda o serão. Assim, o usuário precisa

marcar, e eventualmente empilhar, apenas os nós encontrados ou criados que tenham endereço menor que ECOL.

A variável global ECOL não é necessária; o algoritmo continua funcionando, mesmo que o usuário marque e empilhe todo nó 'inútil' encontrado, em qualquer momento. É difícil comparar a eficiência desta versão com a do algoritmo A6.1; os testes "A<ECOL" deste consomem algum tempo, mas podem diminuir o número de nós marcados em cada ciclo, e, portanto, aproveitar melhor o espaço. O algoritmo A6.1, com os testes de ECOL, tem o mérito de ser mais fácil de analisar.

6.5 Justificativa do Algoritmo de Steele

Para justificar melhor o algoritmo A6.1-A6.2, enunciaremos uma série de afirmações, que serão sempre válidas quando a execução do recuperador atingir certos pontos, desde que nesse momento o processo usuário esteja fora de suas regiões S-críticas.

Recordemos que um nó N é ativo quando for o término de uma cadeia com origem num campo de base. Os nós que estão ativos num dado instante podem estar marcados como 'úteis', 'inúteis' ou 'relocados', dependendo do ponto em que se encontra a execução do recuperador. Os nós da lista livre não são considerados ativos.

Diremos que um nó X é marcável a partir de um campo A se X for o término de uma cadeia de nós, todos com marca = 'inútil' (incluindo X), com origem em A. Definimos então os predicados:

$X \text{ é útil}$	$\equiv X.\text{marca} = \text{'útil'}$
$X \text{ é PE-marcável}$	$\equiv X \text{ é marcável a partir de um campo de um nó cujo endereço está em PE;}$
$X \text{ é R(J)-marcável}$	$\equiv X \text{ é marcável a partir de um campo de base } R[K], \text{ com } J \leq K \leq C_B.$
$X \text{ é E(J)-marcável}$	$\equiv X \text{ é marcável a partir de algum campo } E+.corpo[K] \text{ com } J \leq K \leq \bar{c}.$
$X \text{ foi relocado}$	$\equiv X.\text{marca} = \text{'relocado'}, \text{ e } X.\text{corpo}[1] \text{ designa um nó com marca = 'útil';}$
$X \text{ é ECOL-marcável}$	$\equiv X \text{ é marcável a partir de um campo de algum nó ativo com endereço menor que ECOL.}$

Podemos então afirmar que, nos pontos assinalados do recuperador, valem as seguintes condições, para todo nó ativo X :

- $\{M_0\} \quad X \text{ é R(1)-marcável;}$
- $\{M_1\} \quad X \text{ é útil, ou } X \text{ é R(J)-marcável, ou } X \text{ é PE-marcável;}$
- $\{M_2\} \quad X \text{ é útil, ou } X \text{ é PE-marcável;}$
- $\{M_3\} \quad X \text{ é útil, ou } X \text{ é PE-marcável, ou } X \text{ é E(J)-marcável;}$
- $\{M_4\} \quad X \text{ é útil.}$
- $\{R_{1-3}, A_{1-3}\} \quad X \text{ é útil, ou } X \text{ foi relocado.}$
- $\{C_1\} \quad X \text{ é útil, ou } X \text{ é R(1)-marcável, ou } X \text{ é ECOL-marcável;}$

Além dessas afirmações, podemos acrescentar que, nos pontos $\{R_3, A_{1-3} \text{ e } C_1\}$, valem sempre

- $\{R_3\}$ $E+.marca='útil'$, e $EL+.marca='inútil'$.
- $\{A_1\}$ todo campo de base não atômico $R[K]$, com $1 \leq K < J$, aponta para um nó com $marca='útil'$.
- $\{A_2\}$ todo campo não atômico, na base e em nós com $marca='útil'$ de endereço menor que E , designa um nó com $marca='útil'$.
- $\{A_3\}$ todo campo não atômico, na base e em nós com $marca='útil'$ de endereço menor que E , bem como todo campo não atômico $E+.corpo[K]$ com $1 \leq K < J$, aponta para um nó com $marca='útil'$.
- $\{C_1\}$ todo nó ativo de endereço menor que $ECOL$ tem $marca='útil'$, e nenhum nó com endereço maior ou igual a $ECOL$ tem $marca='útil'$.

Mais ainda, em todos esses pontos podemos acrescentar que nós com $marca='livre'$ estão na lista livre, e que, exceto em $\{M_{1-3}\}$, pilha PE está vazia.

Se $\{a\}$ é um dos pontos assinalados do recuperador, definiremos a α -asserção como sendo a conjunção de todas as afirmações da lista acima associadas ao ponto $\{a\}$.

Definiremos também um α -evento como sendo um instante em que a afirmação "o recuperador encontra-se no ponto $\{a\}$, e o usuário está fora de suas regiões S -críticas" muda de falsa para verdadeira.

Note-se que, como os pontos indicados estão todos no início de regiões S -críticas, sempre que o recuperador atin

ge um ponto $\{\alpha\}$, ocorre pelo menos um α -evento antes que ele possa continuar.

Para provar que, em todo β -evento, a β -asserção correspondente é válida, devemos considerar todos os α -eventos que podem preceder imediatamente um β -evento, e provar que as instruções que podem vir a ser executadas entre ambos, pelo recuperador e pelo usuário, intercaladas em qualquer ordem possível, transformam a α -asserção na β -asserção.

As operações do usuário que não afetarem variáveis globais não tem nenhum efeito sobre as afirmações dadas acima, e podem ser ignoradas nas demonstrações; precisamos apenas nos preocupar com as ações do recuperador, e as seções críticas do usuário.

As seções críticas do usuário podem ser executadas em paralelo apenas com operações do recuperador que não consultam ou modificam variáveis globais. A ordem em que estas operações são intercaladas com as do usuário é totalmente irrelevante, e podemos escolher a ordem mais adequada à demonstração.

Portanto, basta considerar a seguinte sequência de instruções, entre dois eventos sucessivos:

- a) uma seção crítica do usuário, ou
- b) uma seção crítica do recuperador,
seguida de operações locais do recuperador,
seguida de zero ou mais seções críticas do usuário.

Pode-se verificar que a execução completa de qualquer seção crítica do usuário não falseia nenhuma das α -asserções. Pode-se também verificar que a execução de uma se-

ção crítica do recuperador, e das instruções locais compreendidas entre esta e a seção crítica seguinte, transformam a asserção válida antes da primeira na asserção válida antes da segunda, se a execução simultânea do processo usuário for ignorada.

Em vista disso, e do que foi dito anteriormente, conclui-se que, se num α -evento vale a α -asserção correspondente, em todo β -evento que se segue valerá a respectiva β -asserção. Como a execução inicia-se com um M_0 -evento, e a asserção associada a $\{M_0\}$ é verdadeira nesse instante (pois todos os nós tem $\text{marca} = \text{'inútil'}$), concluímos que em todo α -evento vale a respectiva α -asserção.

A afirmação $\{C_1\}$ garante que nós acessíveis ao usuário não serão recolhidos à lista livre.

Um nó inacessível pode ter $\text{marca} = \text{'útil'}$ no momento em que for examinado por COLETA. Sua marca será substituída por 'inútil' nessa ocasião, e nem o recuperador, nem o usuário poderão torná-la 'útil' novamente. Esse nó será recolhido na COLETA seguinte. Portanto, todo nó inacessível é oportunamente reaproveitado.

A afirmação $\{R_3^1\}$ garante que um nó somente será relocado para uma área 'inútil'.

Se a variável global ECOL for eliminada do algoritmo, as asserções $\{C_1\}$ e $\{M_0\}$ deverão incluir também a alternativa X é PE-marcável, pois os procedimentos do usuário poderão marcar e empilhar nós já durante a coleta.

Naturalmente, é necessário comprovar também que a execução do recuperador não afeta "significativamente" a execu-

ção do programa usuário. Ou seja, se este fosse processado com espaço livre ilimitado, sem recuperação, os comandos executados (exceto os contidos nos procedimentos básicos A6.2) e os dados manipulados (exceto endereços de nós) devem ser os mesmos que são executados e manipulados quando o recuperador de Steele é processado em paralelo.

As etapas de marcação e coleta não perturbam a execução do processo usuário. pois alteram apenas as marcas e os nós inativos, que não são por ele consultados fora dos procedimentos básicos. A etapa de relocação pode mover um nó ativo para outro endereço; porém, todo acesso ao mesmo é automaticamente substituído, pelas rotinas básicas, por um acesso à cópia, de modo que o usuário obtém nessa operação um resultado equivalente ao que obteria sem relocação.

Na verdade, se substituirmos idealmente todo apontador para um nó com `marca='relocado'` pelo primeiro campo desse nó, num momento em que os dois processos estiverem fora de regiões críticas, cada campo de base apontará para uma estrutura "equivalente" (como definido na seção 4.3) à que ele apontaria, nesse mesmo instante, se não houvesse recuperação simultânea. A função da etapa de atualização, e das invocações de ATUALIZA nas rotinas de A6.2, é exatamente o de concretizar essa "substituição ideal" sempre que ela for necessária.

As etapas de relocação e atualização podem ser superfluas, considerando-se que os nós tem tamanho uniforme. As duas podem ser eliminadas sem problemas, juntamente com o procedimento ATUALIZA e todas as suas invocações. Note-se que, neste caso, os procedimentos EXTRAI-CAMPO e ALTERA-CAMPO reduzem-se a uma aplicação de ATRIBUI aos campos envolvidos, e que a comparação de dois campos deixa de exi-

gir precauções especiais.

O recuperador resultante foi descrito e analisado por Philip L. Wadler em {Wad 9/76}, após o artigo de Steele, mas é, provavelmente, anterior a este último.

Na discussão do algoritmo de Steele, pressupõe-se que o número de nós ativos (sem contar os 'relocados'), em cada invocação do ALOCADOR, é sempre menor que N . Se o usuário tentar criar um novo nó quando todos os N estiverem ativos, a lista livre estará vazia, e o ALOCADOR ficará preso na malha ESPERA, aguardando em vão que o recuperador colete algum nó.

Numa aplicação prática, é necessário detectar esta situação e cancelar a execução do processo usuário, por falta de espaço. Esta decisão pode ser tomada quando dois ciclos consecutivos do recuperador terminam com a lista livre vazia e com o usuário preso na malha ESPERA.

6.6 Análise do Recuperador de Steele

O objetivo da recuperação em tempo real é evitar pausas demoradas no decorrer da execução do programa do usuário. Se comparada com a execução ideal (com espaço livre ilimitado, sem recuperação), a execução simultânea com o algoritmo de Steele é mais lenta. Essa demora adicional pode ser devida a três causas:

- (1) conflitos entre regiões S-críticas dos dois processos, que causam a suspensão do usuário;
- (2) instruções extras executadas pelo usuário, para compatibilizá-lo com o recuperador, incluídas nas

rotinas ATRIBUI, EXTRAÍ-CAMPO, etc.

- (3) iterações efetuadas pelo alocador (malha ESPERA), para aguardar que o recuperador recolha algum nó à lista livre, se a mesma for encontrada vazia.

Se os dois processos tiverem que compartilhar um único processador, o item (1) acima é substituído por

- (1') suspensões do usuário durante a execução do recuperador.

As pausas mais preocupantes são as devidas ao item (3). Se a lista livre ficar vazia, o processo usuário pode ficar preso em ESPERA durante dois ciclos quase completos do recuperador (praticamente o dobro da pausa gerada pelo algoritmo não paralelo A3.1), e essa interrupção pode se repetir a cada nó criado. É possível eliminar tais pausas, se o espaço total N for suficientemente grande, e se os critérios de prioridade permitirem que o recuperador avance a uma velocidade adequada, de modo que, ao fim de cada ciclo, este último consiga recolher à lista livre nós suficientes para alimentar o alocador durante todo o ciclo seguinte.

Nesta seção, determinaremos condições suficientes para que esse equilíbrio seja mantido.

No início do i -ésimo ciclo do RECUPERADOR-STEELE, sejam

N_L^i = número de nós na lista livre

N_A^i = número de nós acessíveis.

Ao longo desse ciclo, teremos

N_E^i = número de nós empilhados em PE durante a marcação, e

N_X^i = número de nós alocados pelo usuário, no decorrer do ciclo i .

A coleta recolherá à lista livre todo nó que tiver $\text{marca} = \text{'inútil'}$ ou 'relocado' . O número desses nós é igual ao número de nós 'inúteis' no fim da marcação, e portanto $N - N_L^i - N_E^i$.

Assim, no início do ciclo $i+1$, a lista livre conterá

$N_L^{i+1} = N_L - N_X^i + (N - N_L^i - N_E^i) = N - N_X^i - N_E^i$ nós. Como apenas nós que sejam ativos no início da marcação podem ser empilhados em PE, temos $N_E^i \leq N_A^i$, e portanto

$$N_L^{i+1} \geq N - N_X^i - N_A^i. \quad (1)$$

Vamos supor que os critérios de prioridade são tais que, durante a etapa de marcação, pelo menos k_M campos são examinados pelo recuperador, antes de cada invocação do ALOCADOR pelo usuário. Vamos supor ainda que pelo menos k_R e k_C nós tem suas marcas examinadas, por RELOCA e COLETA, respectivamente, e que pelo menos k_B campos de base ou k_A nós são tratados por ATUALIZA-NÓS, antes de cada alocação que o usuário efetue durante essas etapas.

Isso garante que

$$N_X^i \leq (C_B + \hat{C} \cdot N_E^i) / k_M + N / k_R + C_B / k_B + N / k_A + N / k_C \quad (2)$$

Se $N_L^{i+1} > N_X^{i+1}$, o alocador nunca encontrará a lista livre vazia no decorrer do ciclo $i+1$. Lembrando que $N_A^i \geq N_E^i$, de (1) e (2) obtemos uma condição suficiente para isso:

$$N[1-2(1/k_R+1/k_A+1/k_C)] > N_A^i + \hat{c}(N_A^i + N_A^{i+1})/k_M + 2C_B(1/k_M+1/k_B) \quad (3)$$

Se supusermos $1/k_R+1/k_A+1/k_C < 1/2$, e se o número de nós atingíveis, em qualquer instante, for limitado por \hat{N}_A , a condição acima é garantida quando

$$N > \frac{\hat{N}_A(1+2\hat{c}/k_M)+2C_B(1/k_M+1/k_B)}{1-2(1/k_R+1/k_A+1/k_C)} \quad (4)$$

A condição acima pode ser usada para escolher N, k_M, k_R, k_B, k_A e k_C , conhecido \hat{N}_A . Se as etapas de relocação e atualização forem eliminadas do algoritmo, as análises acima continuam válidas, bastando omitir os termos $1/k_R$, $1/k_A$ e $1/k_B$ das fórmulas.

Como é de se esperar, quanto maiores forem k_M, k_R, k_B, k_A e k_C , menor precisa ser N/\hat{N}_A . O aumento desses parâmetros, entretanto, aumenta a duração das pausas devidas a conflitos entre os dois processos sobre o semáforo S , ou sobre o uso do processador.

Seja t o tempo de execução máximo que o recuperador pode precisar, entre duas alocações consecutivas, para conseguir o progresso especificado pelos parâmetros k_X . Seja t_c a porção desse tempo t consumida dentro de seções S -críticas.

Se o usuário e o recuperador compartilharem o mes-

mo processador, a duração total das suspensões do primeiro, entre duas alocações, pode chegar a t . Se dois processadores independentes forem usados, os conflitos ainda podem causar suspensões do usuário com duração total t_c , a cada alocação.

Em termos macroscópicos, essas pausas equivalem a a longar de t ou t_c , conforme o caso, o tempo consumido pelo procedimento ALOCADOR. Como este tem que obedecer, em geral, a limites impostos pela aplicação, o tempo t , e portanto os parâmetros k_x , também são limitados. Estes, por sua vez, determinam pela equação (4) um tamanho adequado para N , dado o limite \hat{N}_A .

6.7 O Algoritmo de Baker

Outro recuperador em tempo real foi proposto por Henry G. Baker Jr., em {Bak 4/78}. Essencialmente, trata-se de um recuperador por cópia, baseado no algoritmo de Cheney (A5.2 sem a fila explícita).

O paralelismo é introduzido nesse recuperador usando técnicas semelhantes às do algoritmo de Steele. Como naquele algoritmo, o usuário deve se preocupar com a possibilidade de encontrar apontadores para nós que foram relocados para outras áreas, e deve colaborar com o recuperador ao modificar apontadores.

Durante a cópia, os nós acessíveis ao usuário estarão espalhados pelos dois semi-espacos. Para que o processo usuário possa ser executado em paralelo, é necessário que este possa determinar o semi-espaço para onde cada campo aponta. Portanto, ao contrário do que foi suposto no capítulo V,

admitiremos que áreas em semi-espacos distintos tem endereços distintos, e que o operador " \downarrow " pode ser aplicado a qualquer apontador. Suporemos também que o teste " ϵ aponta para o espaço novo" é aplicável a todo endereço ϵ , com significado óbvio.

O uso de endereços distintos permite eliminar a marca em cada nó, usada pelo algoritmo de Cheney para distinguir os nós já copiados: estes se caracterizam por estarem no espaço velho, e por terem, no primeiro campo, um apontador para o espaço novo.

No algoritmo (A6.3), supomos que INI e LIM são o endereço e o limite do semi-espaço antigo, e que ININOV, LIMNOV são os do novo. Vamos considerar, inicialmente, o caso de nós não-sobrepostos, auto-descritos, com campos de tamanho uniforme.

O procedimento MOVE-E-ATUALIZA, usado por ambos os processos, ao encontrar um apontador para um nó no espaço antigo, move este para o espaço novo (se ainda não o tiver sido), e atualiza o apontador, fazendo-o designar a nova área. Os nós são copiados para posições consecutivas, no início do espaço novo; ELIV indica a primeira posição livre deste.

No início de cada ciclo, o recuperador aplica MOVE-E-ATUALIZA a todos os campos de base. Em seguida, examina todos os nós que foram copiados, entre ININOV e ELIV, e aplica MOVE-E-ATUALIZA aos seus campos. Durante essa operação, mais nós podem ser copiados, aumentando ELIV.

Eventualmente, todos os nós entre ININOV e ELIV terão sido examinados. Nesse momento, todos os nós ativos estarão no semi-espaço novo, e o conteúdo do antigo pode ser destruído. O recuperador inicia então um novo ciclo, invertendo o papel dos dois semi-espacos.

```

modo nō: [c: inteiro de 1 a  $\hat{c}$ ,
          corpo: sequência de c campos];
áreas ELIV: endereço de nō, inicialmente ININOV,
          EALC: endereço de nō, inicialmente LIMNOV,
          S: semáforo, inicialmente 'livre';

procedimento MOVE-E-ATUALIZA:
    recebendo área A: campo,
    efetua
        se A é apontador,
        e A não aponta para o espaço novo, então
            se A+.corpo[1] aponta para o espaço novo, então
                se ELIV + tam(A) > EALC, então
                    erro {espaço esgotado - impossível prosseguir};
                mova A+ para ELIV+;
                A+.corpo[1] ← ELIV;
                ELIV ← ELIV + tam(ELIV+);
            A ← A+.corpo[1]

```

Algoritmo A6.3 - O recuperador de Baker. (continua)

procedimento RECUPERADOR-BAKER:

efetua

```

[ área E: endereço de nó,
  EC: inteiro de 1 a  $\hat{c}$ ,
  ET: inteiro de 1 a  $\hat{m}$ ;
  repete indefinidamente
  {B0}
  repete, para J desde 1 até CB,
  {B1}
  reserva S;
  MOVE-E-ATUALIZA(R[J]);
  libera S;
  E ← ININOV;
  MALHA: repete indefinidamente
  {B2}
  reserva S;
  se E ≥ ELIV, então
  [ libera S;
    termina MALHA;
  EC ← E+.c; ET ← tam(E+);
  libera S;
  repete, para J desde 1 até EC,
  {B3}
  reserva S;
  MOVE-E-ATUALIZA(E+.corpo[J]);
  libera S;
  E ← E + ET;
  {B4}
  reserva S;
  INI ↔ ININOV;
  LIM ↔ LIMNOV;
  EALC ← LIMNOV;
  ELIV ← ININOV;
  libera S;

```

Algoritmo A6.3 (conclusão)

Após atribuir um endereço β a um campo α , o usuário deve aplicar `MOVE-E-ATUALIZA` a este, pois pode acontecer que β ainda seja um endereço do espaço antigo, e que α já tenha sido examinado pelo recuperador. Se o usuário não tomasse essa precaução, o campo α permaneceria apontando para a área original, mesmo depois da mesma ser reaproveitada no ciclo seguinte.

Além disso, o usuário deve evitar a consulta ou atribuição a campos do nó designado por um apontador α , se α estiver no espaço antigo e tiver sido copiado; neste caso, deve ser usada, em seu lugar, a cópia de α . Mesmo que α não tenha sido copiado, deve-se tomar cuidado para não atribuir a α .corpo[1] um apontador já atualizado, que mais tarde poderia ser tomado pelo endereço da cópia (inexistente) de α . A aplicação de `MOVE-E-ATUALIZA` a α , antes da consulta, resolve ambos os problemas.

As precauções acima serão observadas se o usuário invocar os procedimentos `ATRIBUI`, `ALTERA-CAMPO` e `EXTRAI-CAMPO` (A6.4) para executar atribuições entre campos de base, modificações de nós, e consultas a nós, respectivamente. Como no algoritmo de Steele, a comparação de dois apontadores requer a atualização preliminar dos mesmos; isso é feito pelo procedimento `COMPARA-ENDEREÇOS`.

Na atribuição de átomos de e para campos de base, ou na comparação destes com átomos, basta envolver a operação com reserva S e libera S , para evitar acessos simultâneos.

A cada invocação do `ALOCADOR`, um novo nó é criado, diretamente no semi-espaço novo. Como os campos do nó são inicializados com 'nada', e `ALTERA-CAMPO` atualiza os que são modificados, o relocador não precisa examinar os nós criados durante o ciclo corrente.

procedimento ALOCADOR:

recebendo EC: inteiro de 1 a \hat{c} {número de campos do nó a criar} ,
 e área A: campo {de base, onde colocar o endereço do mesmo},
 efetua

```

[ área ET: inteiro de 1 a  $\hat{m}$ ;
  ET  $\leftarrow$  tam(inteiro de 1 a  $\hat{c}$ ) + EC.tam(campo);
  reserva S;
  ESPERA: repete, enquanto EALC - ET  $\leq$  ELIV,
    [ libera S;
      reserva S;
    EALC  $\leftarrow$  EALC - ET; A  $\leftarrow$  EALC;
    A $\downarrow$ .c  $\leftarrow$  EC;
    repete, para J desde 1 até EC,
      [ A $\downarrow$ .corpo[J]  $\leftarrow$  'nada';
    libera S;
  ]

```

Algoritmo A6.4 - Rotinas do processo usuário, compatíveis
 com o recuperador de Baker.

```

procedimento ATRIBUI:
  recebendo áreas A,B: campos {de base},
  efetua
    [reserva S;
     A ← B;
     MOVE-E-ATUALIZA(A);
     libera S;
  ]

```

```

procedimento EXTRAI-CAMPO:
  recebendo J: inteiro de 1 a  $\hat{c}$ ,
  e áreas A,B: campos {de base},
  efetua
    [reserva S;
     MOVE-E-ATUALIZA(B);
     A ← B+.corpo[J];
     MOVE-E-ATUALIZA(A);
     libera S;
    ]

```

```

procedimento ALTERA-CAMPO:
  recebendo J: inteiro de 1 a  $\hat{c}$ ,
  e áreas A,B: campos {de base},
  efetua
    [reserva S;
     MOVE-E-ATUALIZA(A);
     A+.corpo[J] ← B;
     MOVE-E-ATUALIZA(A+.corpo[J]);
     libera S;
    ]

```

```

procedimento COMPARA-ENDEREÇOS:
  recebendo áreas A,B: campos {de base},
                                RES: booleano {resultado da comparação},
  efetua
    [reserva S;
     MOVE-E-ATUALIZA(A);
     MOVE-E-ATUALIZA(B);
     RES ← A = B;
     libera S;
    ]

```

Algoritmo A6.4 (conclusão)

Portanto, os mesmos são criados em posições consecutivas no fim do semi-espço, em vez de no início deste. O apontador EALC designa o nó criado mais recentemente, e é decrementado a cada alocação. A área de endereço ELIV e limite EALC constitui o espaço livre, disponível para as criações ou relocações de nós que serão efetuadas no decorrer do ciclo.

Caso o alocador não encontre espaço livre suficiente para o nó a ser criado, o processo usuário será suspenso até que o recuperador termine o ciclo corrente e troque os semi-espços. Porém, se MOVE-E-ATUALIZA precisar copiar um nó, e não houver espaço livre para tal, o recuperador será forçado a parar definitivamente, por menor que seja o tamanho total dos nós ativos nesse momento. Esta possibilidade, que não pode ocorrer no recuperador de Steele, é uma desvantagem do algoritmo de Baker.

Para cada ponto indicado com $\{\alpha\}$ no RECUPERADOR-BAKER, daremos abaixo uma asserção que é válida em todo α -evento. A demonstração dessas asserções não é difícil, usando-se as técnicas vistas na justificação do algoritmo de Steele, e elas contêm os pontos essenciais da verificação formal do recuperador.

Diremos que um campo está atualizado quando for átomo, ou apontar para um nó no semi-espço novo corrente. Sejam as afirmações:

- $\{B_1\}$ todo campo de base $R[K]$, com $1 \leq K < J$ está atualizado;
- $\{B_2\}$ os campos de base, e os campos de todo nó entre ININOV e E, estão atualizados;
- $\{B_3\}$ os campos de base, os de todo nó entre ININOV

e E , e os campos $E + \text{corpo } [K]$ com $1 \leq K < J$, estão atualizados;

$\{B_4\}$ os campos de base, e os de todo nó com $ININOV \leq \text{end}(\alpha) < ELIV$, estão atualizados.

Além dessas, valem em todos os pontos assinalados as afirmações

$\{B'_{0-4}\}$ todo nó atingível a partir de um campo de base está no semi-espço antigo, ou entre $ININOV$ e $ELIV$, ou entre $EALC$ e $LIMNOV$.

$\{B''_{0-4}\}$ os campos de todo nó entre $EALC$ e $LIMNOV$ estão atualizados.

Finalmente, suponha que todo apontador atingível que designar um nó do espaço antigo já copiado seja substituído pelo endereço da cópia. Com esta substituição, a estrutura apontada por cada campo de base, num dado α -evento, será equivalente à que esse campo apontaria, se o usuário fosse executado sozinho até esse ponto, e se o espaço de alocação fosse suficiente para tal.

Como o usuário realmente efetua essa substituição sempre que necessária, o resultado final de sua execução não será afetado pela execução simultânea do recuperador.

6.8 Análise do Algoritmo de Baker

Como já mencionamos, o algoritmo de Baker pode se ver impossibilitado de continuar, se o espaço livre for esgotado. Mais que no algoritmo de Steele, é importante determinar con-

dições suficientes para que tal não aconteça.

Seja M_C^i o tamanho total dos nós copiados pelo recuperador durante o ciclo i , e seja M_X^i a soma dos tamanhos de todos os nós que o usuário tentou criar durante esse período. Se

$$M/2 > M_C^i + M_X^i \quad (1)$$

o ciclo i será completado com sucesso, e sem que o usuário tenha sido interrompido por falta de espaço.

Vamos supor que os critérios de prioridade garantem que, a cada vez que o usuário tentar alocar um nó de tamanho m durante o ciclo i , o recuperador aplica MOVE-E-ATUALIZA a campos, de base ou de nós, cujo tamanho total é pelo menos $k_B \cdot m$ ou $k_N \cdot m$, respectivamente. Então, teremos que

$$M_X^i \leq M_B/k_B + M_C^i/k_N \quad (2)$$

donde se conclui que, para garantir (1), basta que se tenha

$$M/2 > M_C^i(1 + 1/k_N) + M_B/k_B \quad (3)$$

Pode-se verificar facilmente que apenas nós que sejam ativos no início do ciclo i são copiados no decorrer do mesmo. Portanto, se \hat{M}_A for um limite superior para o tamanho total dos nós ativos, no decorrer do processamento, uma condição suficiente para que o mesmo termine normalmente é

$$M > 2 [\hat{M}_A(1 + 1/k_N) + M_B/k_B] \quad (4)$$

Como no caso do algoritmo de Steele, o progresso do

recuperador, à velocidade implicada por k_N e k_B , pode fazer com que o usuário seja ocasionalmente suspenso. Essas pausas tem o mesmo efeito global que uma demora extra, em cada alocação, proporcional ao tamanho do nó alocado.

6.9 Técnicas para Aumentar o Paralelismo

Tal como foram apresentados, os algoritmos de Steele e de Baker são constituídos quase que exclusivamente de seções S-críticas. Como qualquer manipulação dos campos de base ou das estruturas, pelo programa usuário, é também realizada dentro de seções S-críticas, o resultado é uma elevada probabilidade de conflitos, que pode tornar a execução dos dois processos praticamente sequencial. Veremos nesta seção várias técnicas que podem ser usadas para diminuir a probabilidade de suspensão dos processos.

Uma técnica simples que atinge esse objetivo é usar um semáforo distinto para controlar os acessos a cada nó do espaço de alocação. Desta forma, a probabilidade de conflito devido a acessos simultâneos a esse espaço diminui bastante, pois só ocorrerá quando os dois processos desejarem ter acesso ao mesmo nó. Esse princípio pode ser estendido aos outros objetos globais (campos de base, lista livre, pilha, etc.).

O uso de um semáforo separado para cada nó e cada campo de base pode ser desaconselhável, em muitos casos. Uma alternativa é a que foi proposta por Steele, quando descreveu seu algoritmo: o uso de dois campos seguradores globais, que contêm a cada instante os endereços dos nós a que cada processo deseja ter acesso exclusivo.

Esses campos são manipulados pelo procedimento SEGU

RA (A6.5) que, dados o endereço de um nó e o número (1 ou 2) do processo chamante, verifica se o primeiro já se encontra no campo segurador do outro processo (isto é, está "seguro" por este). Se necessário, SEGURA aguarda até que o endereço seja retirado de lá, para então colocá-lo no campo segurador do processo chamante, e retornar a este. Assim, se cada processo SEGURAR sempre um nó antes de consultá-lo ou alterá-lo, os dois nunca tentarão manipular simultaneamente o mesmo nó.

Esse esquema ainda não é perfeito, pois o semáforo SSEG, que disciplina os acessos aos campos seguradores, torna-se ele próprio uma fonte de conflitos frequentes. Entretanto, as seções SSEG-críticas de SEGURA são extremamente curtas, e portanto as suspensões a ele devidas serão, muito provavelmente, mais raras e menos demoradas que as causadas por S nos algoritmos originais.

Dependendo de detalhes de codificação, é possível que cada processo deseje ter acesso exclusivo a dois ou mais nós ao mesmo tempo. É possível generalizar SEGURA para cobrir esse caso, devendo-se porém tomar cuidado com "impasses" que podem ocorrer. Por exemplo, o usuário "segura" o nó α , o recuperador "segura" β , o usuário tenta também "segurar" β , o recuperador tenta "segurar" α - e os dois processos ficam suspensos permanentemente, cada qual esperando que o outro "solte" um nó, para poder prosseguir.

O procedimento SEGURA pode também ser usado para controlar o acesso a campos de base, e a outras áreas globais, bastando que os campos seguradores possam conter endereços desses objetos, além de endereços de nós.

Os conflitos sobre a lista livre, no algoritmo de Steele, podem ser diminuídos transformando-se a mesma numa

fila, sendo os nós inserido num extremo pelo recuperador, e retirados do outro pelo usuário. Assim, só haverá conflitos quando a fila ficar reduzida a um único nó.

Da mesma forma, a pilha PE pode ser transformada numa dupla fila de saída restrita ({Knu 68 - 2.2.1}), com o usuário inserindo endereços num extremo, e o recuperador retirando-os e inserindo-os pelo outro.

Nestas aplicações, é conveniente generalizar o conceito de semáforo, de modo que seu conteúdo seja um número inteiro entre 0 e N, em vez de apenas 'livre' ou 'ocupado'. A operação libera S soma 1 a esse conteúdo, e reserva S subtrai 1 do mesmo, suspendendo o processo que a executou se e enquanto o conteúdo de S for nulo.

A sincronização desejada para a lista livre usa tal semáforo, cujo conteúdo a cada instante é um a menos que o número de nós na lista. O usuário executa reserva S ao alocar cada nó, e o recuperador libera S ao recolher um. Assim, o primeiro será suspenso apenas quando a lista livre contiver um único nó.

A técnica mais promissora, entretanto, parece ser o abandono completo do uso de semáforos, e basear o controle mútuo dos dois processos apenas na sincronização natural dos acessos à memória que existe na maioria dos sistemas.

Essa sincronização natural geralmente garante que as operações de consulta ou alterações de uma área suficientemente pequena (um campo, ou um item de um nó) são indivisíveis, e mutuamente exclusivas. Isto, é,

- (1) Se dois processos tentam consultar simultaneamente o mesmo item, o resultado das duas con

sultas é o conteúdo corrente do mesmo.

- (2) Se um processo consulta um item que o outro está alterando, a alteração é efetuada corretamente, e o resultado da consulta é o conteúdo original do item, ou o que lhe foi atribuído.
- (3) Se dois processos tentam alterar ao mesmo tempo um item, o conteúdo final deste é um dos dois valores que lhe foram atribuídos.
- (4) Se dois processos tentam manipular dois itens disjuntos, do mesmo nó ou de nós distintos, as duas operações são efetuadas corretamente, sem interferência mútua.

Se essas propriedades forem satisfeitas, é possível reescrever o algoritmo de Steele (e, provavelmente, também o de Baker) eliminando totalmente semáforos e seções críticas. Essa técnica foi primeiro apresentada, aparentemente, por David Gries ([Gri 12/77]), que a aplicou a um recuperador, simples mas ineficiente, devido a E.W.Dijkstra.

Uma descrição completa do algoritmo de Steele (sem compactação) que dispensa totalmente o uso de semáforos, acompanhada de uma demonstração rigorosa de sua correção, foi apresentada por Siang W.Song e H.T.Kung em {Kun/Son 5/77}.

Com esta técnica, o paralelismo é virtualmente completo, pois só haverá conflito quando os dois processos tentarem manipular o mesmo item simultaneamente. E, neste caso, a suspensão não durará mais que o tempo necessário para uma consulta ou atribuição.

CAPÍTULO VII - CONCLUSÕES FINAIS

7.1 .Comparação dos Métodos de Recuperação

A escolha do método de recuperação mais adequado depende fortemente das características da aplicação considerada. Para cada um dos métodos vistos neste trabalho, existe uma situação prática para a qual ele é o mais indicado.

Os métodos de coleta simples, com lista livre, são geralmente adequados a aplicações com nós de tamanho uniforme, sem memória paginada, devido à sua simplicidade e relativa eficiência.

Recuperadores baseados em cópia ou compactação são praticamente indispensáveis em sistemas com memória paginada, ou com nós de tamanho variado. O algoritmo de Edwards modificado (A4.8), e os copiadores de Cheney e Clark, são também aconselháveis quando a parte ativa do espaço de alocação é relativamente pequena, pois seu tempo de execução depende apenas do tamanho desta.

Apesar de sua simplicidade, os algoritmos de cópia perdem para os de compactação no que se refere ao aproveitamento do espaço, à generalidade, e até mesmo à eficiência. Os copiadores de Cheney e Clark são vantajosos apenas no caso de nós não superpostos, suficientemente grandes e de tamanho variado; se os nós tiverem tamanho uniforme, o algoritmo de Edwards ou o A6.1 são sérios concorrentes. No caso de nós superpostos (ou muito pequenos), apenas o algoritmo de Zave (A4.10 adaptado) e o baseado na marcação previa dos segmentos

ativos são aplicáveis, e não são visivelmente superiores ao compactador de Morris.

Da mesma forma, em aplicações de tempo real, o algoritmo de Steele/Wadler é preferível para nós de tamanho uniforme, enquanto que o de Baker é mais adequado a nós não-superpostos de tamanho variado. Para nós superpostos, ou muito pequenos, é possível adaptar o compactador com marcação prévia.

7.2 Pesquisas em Andamento e Problemas em Aberto

A construção e análise de algoritmos recuperadores é uma das áreas de pesquisa em ciência da computação que está mais ativa no momento. Embora grandes progressos tenham sido realizados desde a proposta original de McCarthy, são muitos os problemas que ainda estão em aberto.

Um exemplo de tais problemas, de grande relevância prática, é uma combinação mais íntima das técnicas descritas neste capítulo com a alocação por pilha, geralmente usada para o espaço de base.

A motivação para esse problema é a constatação de que a disciplina de pilha usada na criação e destruição dos campos de base, e a redução dos efeitos colaterais de procedimentos induzida pelas técnicas de programação estruturada, fazem com que os nós mais recentemente criados sejam, com frequência, os primeiros a serem destruídos.

Um algoritmo recuperador que levasse esse fato em consideração, auxiliado por informações sobre a estrutura do programa usuário (recolhidas pelo compilador, ou garantidas pelas regras do sistema), poderia realizar a coleta ou compactação das estruturas em tempo muito menor que o de um algoritmo convencional.

Algoritmos com estas características são propostos e estudados, por exemplo, por D.M.Berry e A.Sorkin {Ber/Sor 4/78}.

Um campo que praticamente não foi explorado é a otimização dos recuperadores em sistemas de memória paginada. Pode-se sugerir o desenvolvimento de recuperadores, executados simultaneamente com o programa usuário, que evitem gerar falhas de páginas adicionais, aproveitando ao máximo as transferên-

cias de páginas causadas pelo próprio usuário. Ou, ainda, algoritmos eficientes que redistribuam os nós ativos entre as várias páginas, de modo a minimizar as falhas do usuário. O artigo {Bae 11/72} é um dos trabalhos nesta área.

A compactação de estruturas é outro tópico em que progressos seriam altamente desejáveis. Um compactador, com tempo linear em M_A , que fosse aplicável a nós de tamanhos variados ou muito pequenos, teria aplicações imediatas.

Num estudo que já mencionamos {Cla/Gre 2/77}, D. W. Clark e C.C.Green mostraram que, na linguagem LISP1.5, os filhos de um nó geralmente estão próximos a ele. Sugeriram portanto que todo campo apontador α fosse substituído por uma área, substancialmente menor, na qual seria armazenada a diferença $\alpha - \text{end}(\alpha)$. Nos poucos casos em que essa área fosse insuficiente, usar-se-ia o método de "ligação por tabela" devido a D.G.Bobrow {Bob 7/75}.

Essa técnica economizaria uma quantidade substancial de memória. Entretanto, é necessário rever e adaptar muitos dos algoritmos recuperadores para que funcionem eficientemente com essa representação. Outro problema que surge nesse contexto é o desenvolvimento de algoritmos compactadores que aumentem essa "localidade" dos apontadores (note-se que o algoritmo de Edwards, A4.7, tende a destruir tal propriedade).

Nesse mesmo estudo, Clark e Green observaram que a maioria dos nós tem apenas um pai. Isso motivou uma proposta, por L.P.Deutsch e D.G.Bobrow {Deu/Bob 9/76}, de combinar um algoritmo recuperador dos anteriormente estudados com uma técnica mais antiga, baseada em contadores de referências.

Nesta técnica, guarda-se para cada nó (num item do mesmo, ou numa tabela separada) o número de apontadores que

o designam. Este número é atualizado sempre que um campo apontador é alterado, ou destruído; quando atinge zero, sabe-se com certeza que o nó é inacessível, e o mesmo pode ser recolhido à lista livre.

A técnica de recuperação por contadores de referências é extremamente ineficiente, por exigir $N \lceil \log_2 N \rceil$ bits para os contadores, e tempo considerável para sua atualização. Além disso, é imperfeita, pois nós inativos que pertencerem a uma cadeia fechada nunca recolhidos.

A proposta de Deutsch e Bobrow consiste em manter os contadores em duas tabelas, com entradas apenas para nós com zero ou com mais de um antepassados imediatos. Em vista dos resultados de Clark e Green, a maioria dos nós não será representada em nenhuma das duas tabelas, o que diminui consideravelmente o espaço ocupado pelas mesmas. Ainda haverá necessidade de um algoritmo recuperador, como os que estudamos, para cuidar de estruturas com cadeias fechadas, nas suas invocações serão bem mais raras, e o artigo de Deutsch e Bobrow sugere várias técnicas que podem ser usadas em sua otimização.

Na linguagem ALGOL 68, a parte ativa de um nó pode ser constituída de duas ou mais áreas não consecutivas da memória; é o caso de slices de matrizes. Essas áreas podem ser relocadas e compactadas, desde que os endereços dos elementos de um slice sejam sempre dados por uma fórmula linear em seus índices.

Como os compactadores dos capítulos 4 e 5 não conservam essa propriedade, a recuperação de espaço em ALGOL 68 é feita tratando-se as áreas entre os elementos de um slice como parte do registro do mesmo. Isso pode acarretar desperdícios graves de memória; não conhecemos, entretanto, nenhum algoritmo compactador que alivie esse problema.

Outro problema em aberto, de grande relevância prática, é a construção de recuperadores incrementais, isto é, cujas invocações, ao longo de todo o processamento do programa usuário, consomem no máximo uma fração fixa do tempo consumido por este. Os algoritmos de tempo real, vistos no capítulo 6, representam uma tentativa neste sentido; entretanto, essa fração é relativamente grande, e depende da relação M_A/M . Se possível, um recuperador que exigisse tempo reduzido, e proporcional ao número de alterações de apontadores efetuadas desde sua invocação anterior, seria altamente desejável.

Finalmente, podemos observar que nenhum dos métodos vistos neste trabalho é demonstradamente ótimo, de modo que para cada um apresenta-se o problema de melhorá-lo, ou provar que isto é impossível. Em vista das melhorias surpreendentes que surgiram, e continuam a surgir, em algoritmos considerados "ótimos", a primeira alternativa parece ser a mais provável.

O resultado de todos esses avanços recentes nos algoritmos de recuperação tornaram a alocação dinâmica com disciplina liberal uma das técnicas fundamentais da programação de computadores. Apesar dos preconceitos contra ela, motivados pela ineficiência de suas primeiras implementações, essa técnica está sendo rapidamente difundida. Por um lado, a linguagem LISP 1.5 continua sendo cada vez mais usada e refinada; por outro, a alocação dinâmica liberal foi incluída em virtualmente todas as linguagens modernas de programação.

Essa técnica foi inclusive incorporada à própria arquitetura de diversos computadores, concretos e projetados, de modo que, num futuro próximo, o próprio conceito "clássico" de memória, linear e imutável, pode vir a ser substituído pelo de uma coleção de nós, "criados" e "destruídos" à von

tade do programador. Como um dos pesquisadores da área observou, a possibilidade de algum dia usarmos LISP para controlar o fogão da cozinha pode não ser tão absurda quanto parece.

BIBLIOGRAFIA

{Bae 11/72}

Baecker, H.D. - Gabarge collection for virtual memory computer systems. CACM 15,11(72)981-986.

{Bak 4/78}

Baker Jr., H.G. - List Processing in real time on a serial computer. CACM 21,4(78)280-294.

{Ber/Sor 4/78}

Berry, D.M.; Sorkin, A. - Time required for garbage collection in retention block-structured languages. Int. J. Comp. Inf. Sci. 7,4(78)361-404.

{Bob 7/75}

Bobrow, D.G. - A note on hash linking. CACM 18,7(78)413-415.

{Cam 11/74}

Campbell, J.A. - Optimal use of storage in a simple model of garbage collection. Info Proc. Letters 3,2(74)37-38.

{Car 4/60}

McCarthy, J. - Recursive functions of symbolic expressions and their computation by machine, part I. CACM 3,4(60)184-195.

{Che 11/70}

Cheney, C.J. - A non-recursive list compacting algorithm. CACM 13,11(70)677-678.

{Cla 6/76}

Clark, D.W. - An efficient list-moving algorithm using constant workspace. CACM 19,6(76)352-354.

{Cla/Gre 2/77}

Clark, D.W.; Green, C.C. - An empirical study of list structure in LISP. CACM 20,2(77)78-87.

{Deu/Bob 9/76}

Deutsch, L.P.; Bobrow, D.G. - An efficient, incremental, automatic garbage collector. CACM 19,9(70)522-526.

{Dij 11/68}

Dijkstra, E.W. - The structure of the THE multiprogramming system. CACM 11,5(68).

{Fen/Yoc 11/69}

Fenichel, R.R.; Yochelson, J.C. - A LISP garbage-collector for virtual-memory computer systems. CACM 12,11(69) 611-612.

{Fis 7/74}

Fisher, D.A. - Bounded workspace garbage collection in an address-order preserving list environment. Info. Proc. Letters 3,1(74)29-32.

{Gri 12/77}

Gries, D. - An exercise in proving parallel programs correct. CACM 20,12(77)921-930.

{Hoa 3/74}

Hoare, C.A.R. - Optimization of store size for garbage collection. Info. Proc. Letters 2(74)165-166.

{Had/Wai 6/67}

Haddon, B.K.; Waite, W.M. - A compaction procedure for variable-length storage elements. Comp. J. 10(67)162-165.

{Knu 68}

Knuth, D.E. - The art of computer programming, vol I: fundamental algorithms. Addison-Wesley, Reading, Mass. 1969.

{Knu 73}

Knuth, D. E. - The art of computer sorting and searching. Addison-Wesley, Reading, Mass. 1973. vol III:

{Kun/Son 5/77}

Kung, H.T.; Song, Siang W. - artigo a ser publicado.

{Lin 1/73}

Lindstom, G. - Scanning list structures without stacks or tag bits. Info. Proc. Letters 2(73)47-51.

{Mor 8/78}

Morris, F.L. - A time-and space-efficient garbage compaction algorithm. CACM 21,3(78)662-665.

{Rei 3/73}

Reingold, E.M. - A non-recursive list moving algorithm. CACM 16,5(73)305-307.

{Ric 69}

Richards, M. - BCPL: a tool for compiler writing and systems programming, I. Proc. AFIPS 1969 SJCC, vol 34. AFIPS Press, Montvale, NJ, p. 557.

{Rob 6/77}

Robson, J.M. - A bounded storage algorithm for copying cyclic structures. CACM 20,6(77)431-433.

{Sch/Wai 8/67 }

Schorr, H.; Waite, W.M. - An efficient machine-independent procedure for garbage collection in various list structures. CACM 10,8(67)501-506.

{Ste 9/75}

Steele Jr., G.L. - Multiprocessing compactifying garbage collection. CACM 18,9(75)495-508.

{Wad 9/76}

Wadler, P.L. - Analysis of an algorithm for real-time garbage collection. CACM 19,9(76)491-500.

{Weg 3/72}

Wegbreit, B. - A generalised compactifying garbage collector. Comp. J. 15,3(72)204-208.

{Weg 9/72}

Wegbreit, B. - A space-efficient list structure tracing algorithm. IEEE Trans. Comp. (set. 1972)1009-1010.

{Wei 67}

Weissman, C. - LISP 1.5 primer. Dickenson Publ. Company, Inc. - Belmont, CA(1967).

{Wei 7/69}

Weizenbaum, J. - Recovery of reentrant list structures in SLIP. CACM 12,7(69)370-372.

{Wij 75}

van Wijngaarten, E. et al. - Revised report on the algorithmic language ALGOL 68. Acta Inf. 5,4(75)1.

{Wul/Rus/Hab 12/71}

Wulf, W.A.; Russel, D.B.; Habermann, A.N. - BLISS: a language for systems programming. CACM 14,12(71)780.

{Zav 7/75}

Zave, D.A. - A fast compacting garbage collector. Info. Proc. Letters 3,6(75)167-169.