UNIVERSIDADE FEDERAL DE PERNAMBUCO
DEPARTAMENTO DE ELETRÔNICA E SISTEMAS
MESTRADO PROFISSIONALIZANTE

Pós-Graduação em Engenharia Elétrica

Um Algoritmo para Gerenciamento Consistente de Páginas Web

por

Albanir Silva de França

Recife – PE

Março de 2004

Albanir Silva de França

Um Algoritmo para Gerenciamento Consistente de Páginas Web

Este trabalho foi apresentado à Pós-Graduação em Engenharia Elétrica do Centro de Tecnologias e Geociências da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Engenharia Elétrica.

ORIENTADOR: Prof. Dr. Rafael Dueire Lins

Universidade Federal de Pernambuco

Pós-Graduação em Engenharia Elétrica

PARECER DA COMISSÃO EXAMINADORA DE DEFESA DE DISSERTAÇÃO DE MESTRADO PROFISSIONALIZANTE DE

Albanir Silva de França

TÍTULO

"Um Algoritmo para Gerenciamento Consistente de Páginas Web"

A comissão examinadora composta pelos professores: RAFAEL DUEIRE LINS, DES/UFPE, VALDEMAR CARDOSO DA ROCHA JUNIOR, DES/UFPE E FRANCISCO HERON DE CARVALHO JUNIOR, Escola Politécnica/UPE sob a presidência do primeiro, consideram o candidato **ALBANIR SILVA DE FRANÇA**

APROVADO

Recife, 01 de março de 2004.

RAFAEL DUEIRE LINS

VALDEMAR CARDOSO DA ROCHA JUNIOR

FRANCISCO HERON DE CARVALHO JUNIOR

Dedico ao meu filho
Samuel Silva de França.
Que DEUS o abençõe!

Agradecimentos

Em primeiro lugar a DEUS pela vida e pelas bênçãos.

Ao professor Rafael Dueire Lins, orientador, pela sua inestimável ajuda e perseverança.

Ao meu filho Samuel e à Manuela, por amor e carinho, além de suportarem minha ausência nestes meses. Também ao meu pai Ademar e minha tia mãe Esmeralda, pela criação.

Ao Instituto de Tecnologia da Amazônia - UTAM - pela oportunidade que me concedeu para realizar este curso e à Fundação Centro de Análise, Pesquisa e Inovação Tecnológica - FUCAPI - pela oportunidade de ter concedido tempo para conclusão desta dissertação e pela minha ausência no ambiente de trabalho.

Ao meu amigo Raimundo Barreto e família, pela acolhida aqui em Recife.

Aos meus colegas de trabalho e mestrado, Jucimar, Ricardo, Raimundo e Neide pelo apoio que me deram nessa jornada. Aos demais colegas de trabalho da Fucapi pelo apoio e solidariedade.

Ao Unilton e Felipe pela amizade e pela ajuda nos momentos que precisei.

Aos novos colegas que conheci do curso de Engenharia Elétrica: Cristiane, Rui, Rivaldo, Sampaio e Guedes, pela companhia e momentos de convivência.

Às funcionárias da Secretaria de Pós-Graduação do Departamento de Eletrônica e Sistemas: Andréa e Cybele pela colaboração indispensável na parte burocrática do departamento.

À Universidade Federal de Pernambuco pela contribuição para atingir mais um degrau em minha formação profissional.

Resumo

Com o advento da World Wide Web na década de 90, a Internet ganhou uma nova dimensão jamais imaginada pelo seu criador Tim Berners-Lee e seus colegas do CERN. A Web, como é conhecida hoje, utiliza documentos dinâmicos - denominados hipertextos - interligados entre si por meio dos denominamos hiperlinks.

Os hiperlinks formam um grafo de alta complexidade e conectividade. Hoje, inexiste uma maneira consistente de gerenciar as páginas Web, levando freqüentemente a seguirmos endereços de páginas removidas.

Esta dissertação apresenta uma aplicação do algoritmo de coleta de lixo para o gerenciamento consistente de páginas Web. Este algoritmo é baseado no algoritmo de gerenciamento automático de memória em sistemas distribuídos proposto por Lins, que utiliza a contagem de referência ponderada distribuída, aplicado à identificação e coleta de páginas não mais referenciadas, que podem estar em árvore ou formando ciclos.

Abstract

The last decade of the 20th Century witnessed the birth and widespreading at an enormous rate of the Word Wide Web, reaching dimensions unforeseen by its creators Tim Berners-Lee and his colleagues at CERN.

Hiperlinks connect Web pages forming a highly complex and connected graph. There is no automatic management of Web pages, today, leading browsers to dangling references.

This dissertation presents an application of algorithm for the consistent management of Web pages. This algorithm is based on Lins' algorithm for distributed memory management. The proposed algorithm works properly even in the case of pages forming cycles.

Conteúdo

Lista de Figuras

Lista de Algoritmos

Capítulo 1
Introdução

O grande sucesso da Internet a partir da década de 90, principalmente após a implantação do serviço denominado World Wide Web (ou simplesmente Web), os números de páginas em HTML existentes vêm crescendo muito. Efetuar uma busca na rede mundial de computadores tornou-se uma atividade quase corriqueira. Nesse contexto, surgiram os famosos engenhos de busca. Sem eles é praticamente impossível pesquisarmos qualquer assunto na rede mundial de computadores. Vários algoritmos foram implementados para otimizar essas buscas.

Quanto aos mecanismos de buscas, esses podem ser classificados em dois tipos: os organizados em diretórios e os baseados na busca direta através das ligações (hiperlinks). No primeiro tipo encontram-se o Yahoo **[Yah]** e o Cadê **[Cad]**. Nestes, o cadastro das páginas é feito de forma não automática; são feitas por técnicos de forma direta ou por solicitação de cadastro. No segundo, encontramos o Google **[Goo]** e o Altavista **[AV]**. Estes para realizarem suas pesquisas se baseiam nas informações léxicas contidas nos códigos de hipertextos das páginas Web. Para tanto é necessário que o usuário informe determinada palavra (ou palavras) chave para que o mecanismo de busca execute sua função. Esses mecanismos entram numa página e pesquisam entrando em outras páginas indicadas nos vínculos das ligações, fazendo uma busca completa.

Os engenhos de busca que se baseiam na busca direta na World Wide Web utilizam uma técnica chamada *crawling*, que é uma espécie de coleta de documentos. Para realizar esta atividade, eles utilizam o que se denomina hoje de Robô, que também pode ser chamado de *spider* ou *crawler*. Os Robôs são programas autônomos que procuram referências (ligações) para documentos; realizam o "*download*" desses documentos e depois fazem um processamento local para extrair índices. Sua execução segue de modo recursivo seguindo as ligações de hipertexto encontrados no processamento para depois fazer coleta exaustiva na Web.

O maior problema que existe hoje na World Wide Web é a constante mudança de endereços e as retiradas de "páginas do ar", fazendo com que as referências dessas buscas se

tornem vazias. Os mecanismos de buscas sempre têm que re-atualizar seus dados através de novos rastreamentos de páginas, desperdiçando tempo de processamento e sendo condenados à incerteza da incompletude da tarefa face à grande dinâmica do sistema. Outra conseqüência dessas constantes mudanças é a superlotação dos servidores com páginas sem nenhuma referência interna ou externa, gerando mão de obra ao usuário que gerencia estas páginas no diretório vinculado à Internet. Estas páginas constituem o que podemos denominar de lixo.

Para entendermos melhor o contexto da World Wide Web, vamos rever alguns conceitos relevantes para justificarmos o objetivo deste trabalho. Iniciaremos nosso estudo com a descrição das partes básicas que envolvem a Web, como o hipertexto, a linguagem de formatação e o protocolo de comunicação HTTP.

1.1 Hipertexto

Vamos analisar antes a etimologia da palavra hipertexto: sufixo “hiper” + o substantivo “texto”. O sufixo vem do grego “*hyper*” que significa posição superior, excesso, além **[PU03]**. O substantivo “texto”, segundo o dicionário Michaelis **[DEM]**, é: s. m. 1. Conjunto de palavras, frases escritas. 2. As próprias palavras de um autor, livro ou escrito. 3. Palavras que se citam para provar alguma coisa. 4. Passagem da Escritura que forma o assunto de um sermão.

O texto pode ser visto como um corpo de informação registrada. Nesta abordagem podemos considerar texto e documento como sinônimos, embora este último possa conter imagens (o que contraria a definição geral de texto).

O hipertexto vai além do conceito simples de texto. Enquanto este possui uma dimensão linear (em uma dimensão), o hipertexto pode ter múltiplas dimensões, que inter-relaciona vários textos. Podemos citar como exemplos concretos os livros de referências, como enciclopédias e dicionários.

Um dos grandes problemas de se fazer pesquisa em uma enciclopédia ou uma biblioteca é a busca exaustiva por determinados assuntos de interesse em uma pesquisa. Outro fator negativo é o volume físico e o peso dos materiais. Os sistemas de hipertexto eletrônico

eliminam esses problemas, possibilitando acesso a um banco de dados com um grande volume de informação em uma curta fração de tempo.

Com o surgimento da Internet, o hipertexto passou a ter um significado intrinsecamente associado ao conteúdo de uma página eletrônica. Um sistema de hipertexto passou a ser definido como uma forma de distribuição da informação na qual os dados são armazenados em uma rede de computadores. Com o aparecimento da tecnologia de multimídia e sua vinculação imediata ao hipertexto, surgiu o termo hipermídia, onde textos, som, imagens e vídeos se condensaram em uma página. As ligações (hiperlinks) passaram a interligar não somente computadores, mas também as páginas contidas no mesmo computador ou em computadores remotos.

1.1.1 Referências (hiperlinks) e âncoras

As ligações entre os documentos no hipertexto em meio eletrônico e sua facilidade de utilização tornou-o bastante atraente entre os usuários, vindo a ser posteriormente com o advento da Web mais do que meramente um conjunto de páginas interligadas. Segundo Glushko **[Glu89]**, o usuário pode navegar em um mar de unidades de texto, seguindo referências que estão relacionadas de alguma forma interessante.

Uma referência pode estar relacionada com uma âncora. Uma âncora é uma região da tela que é sensível a qualquer dispositivo apontador que estiver sendo usado, como mouse, caneta ótica, dedo. A ativação, como um clique no mouse, sobre uma âncora faz com que haja um desvio para outra região do texto. Estas ligações permitem ao hipertexto ter uma forma estrutural com determinadas propriedades. Em uma página pode haver vinculações entre pontos no próprio documento, entre documentos externos, entre índices e documentos, entre documento e imagens, vídeos entre outras.

A introdução de vínculos em um sistema de texto comum implica na necessidade de criação de mecanismos para dar suporte a sua criação, alteração, remoção, alteração de nomes e atributos e outras características próprias.

Ligações (hiperlinks) são usualmente denotados por palavras ou frases que são destacadas de modo acessível, mas que também pode ser gráfico ou em forma de ícones. Por

exemplo, cada componente de um diagrama esquemático pode ser um link para um diagrama mais detalhado daquele componente ou de uma outra descrição textual. As ligações podem produzir uma variedade de resultados diferentes. Eles podem **[SK89]**:

- Transferir-se para um novo tópico;
- Mostrar uma referência (ou ir de uma referência para um artigo);
- Fornecer uma informação auxiliar, tais como notas de rodapé, definição ou anotação;
- Exibir uma ilustração, esquema, fotografia ou uma seqüência de vídeo;
- Exibir um índice;
- Executar um programa (tipo uma planilha ou uma animação).

1.1.2 Navegação nos hipertextos

Quando as palavras de um documento são palavras-chaves ou índex de outras palavras que permitam uma pesquisa eficiente, gera-se o que se chama de sistema hipertexto, onde as palavras, parágrafos e pensamentos são interligados, e o usuário pode navegar através do documento ou entre documentos diferentes de uma forma não linear, rápida e intuitiva.

A navegação é ponto chave nos sistemas de hipertexto. Um usuário pode navegar seguindo inúmeras referências e depois de determinado caminho percorrido não saiba mais onde está ou em que ponto se perdeu.

Nos sistemas de hipertexto, a navegação pode ser realizada por meio de busca e por “folheamento” dos documentos. Uma busca pode ser realizada utilizando uma palavra ou um conjunto de palavras chaves sobre alguma informação específica. Já a ação de folhear, consiste no desvio de um documento para outro de forma seqüencial, não necessariamente ordinal. Segundo **[Rad91]**, estes dois tipos de navegação ocorrem em adição à leitura, que é acobertada seqüencial linha a linha de todo um documento.

1.1.3 Um breve histórico do hipertexto

O termo hipertexto foi criado por Ted Nelson (Theodor Holm Nelson) em 1965, em decorrência da idealização de um ambicioso projeto denominado Xanadu[1]. Ele pretendia criar um ambiente literário e de editoração global, mantendo todas as publicações disponíveis para acesso via meio eletrônico. Esse projeto - ainda incompleto, talvez por seu escopo extremamente grandioso, visa criar um *mundo virtual* de documentos (um *docuverse*, segundo Nelson) no maior número possível, organizados de maneira extremamente hipertextual.

Mas antes de Ted Nelson, podemos citar como precursor dos atuais sistemas de hipertexto, o sistema *Memex*, descrito por Vannevar Bush em 1945 **[Bus45]**. No começo da década, com o impulso das necessidades tecnológicas trazidas pela 2ª Guerra Mundial, são desenvolvidos os primeiros computadores. Em 1945, é construído o ENIAC, considerado o primeiro computador eletrônico na história dos computadores. Em julho desse ano, Vannevar Bush publica seu artigo *As we may think* (Como nós podemos pensar) na revista *The Atlantic Monthly* **[TAM45]**. Bush era um cientista do pós-guerra que após muito desenvolvimento tecnológico devido às pesquisas bélicas, buscou motivos para continuar suas pesquisas em novos horizontes. Sua idéia era uma nova maneira de estruturar e organizar o conhecimento, fosse ele pessoal ou universal. Estudou uma maneira de entender o funcionamento do cérebro humano e desenvolveu uma teoria **associativa**. Para defendê-la ele imaginou um dispositivo que denominou *Memex*. Aproveitando-se basicamente de tecnologias já existentes, ele implementa essa maneira associativa de organizar e acessar o conhecimento, em um nível, porém, ainda um tanto pessoal. Além da enorme importância da inovação dessa proposta teórica, esse "*insight*" de Bush adiciona um novo elemento ao surgimento do hipertexto no campo computacional com a introdução da necessidade de interatividade. Enquanto no campo artístico essa questão já estava implícita nas próprias obras, no campo científico essa questão era praticamente intocada. Um leitor de uma enciclopédia não podia adicionar em um verbete uma referência a outro assunto, a menos, é claro, que fizesse anotações.

O *Memex* nunca chegou a ser implementado. Fisicamente, o sistema seria como uma mesa de trabalho com telas de alta resolução, teclado e botões de controle. O armazenamento seria feito utilizando-se microfichas. O acesso a estas seria feito de modo mecânico e,

efetuado através de índices. Dois itens quaisquer dentro do sistema poderiam ser codificados para associação permanente. Esta associação seria chamada trilha, à semelhança da trilha de associação na mente do usuário **[Rad91]**.

1.2 Linguagens de marcação (markup)

Uma linguagem de marcação é definida com sintaxe textual extra que pode ser utilizada para descrever ações de formatação, de informação estruturada, de semânticas de texto, de atributos, etc. O termo "marcação" vem da época em que os editores realmente marcavam os documentos para informar ao impressor que fontes utilizar e assim por diante **[Tan03]**. Linguagens de formatação como o TeX **[Tex]** pode ser considerada de marcação. Entretanto, linguagens de marcação formais são bem mais estruturadas. Os marcadores são chamados etiquetas (*tags*), e usualmente, para evitar ambigüidade, há uma etiqueta inicial e outro final demarcando o texto. A metalinguagem padrão para formatação é a SGML (Standard Generalized Markup Language). A linguagem de marcação mais conhecida e utilizada na Web é o HTML (HyperText Markup Language - *Linguagem de Marcação de Hipertexto*), que é uma instância da SGML **[YN99]**.

1.2.1 SGML – Standard Generalized Markup Language

O SGML foi definido pelo Padrão ISO 8879 como uma metalinguagem para formatação de texto desenvolvido pelo grupo de Golfarb **[Gol90]** inspirado num trabalho anterior desenvolvido na IBM. Ele não foi desenvolvido para hipertexto, mas define regras para uma linguagem de formatação baseada em etiquetas.

Cada instância do SGML inclui a descrição de uma estrutura de documento chamada de Definição de Tipo de Documento (**DTD - Document Type Definition)**. Assim um documento SGML é definido por **[YN99]**:

- Descrição da estrutura do documento;

[1] Xanadu significa "lugar mágico da memória literária" e foi retirado do poema *Kubla Khan* de Samuel Taylor Coleridge [CON87].

- Formatação do próprio texto com tags que descrevem a estrutura.

O HTML é definido segundo um DTD de SGML e algumas das suas principais características são:

- Formatação de documentos;
- Organização de listas;
- Suporte a hipertexto/hipermídia em documentos Web;
- Suporte à imagens clicáveis (ícones).

1.2.2 HyTime – Hypermedia/Time-based Document Structuring Language

O Hy Time (ISO/IEC 10744 - 1992) é um padrão definido para representação estruturada de hipermídia e informação baseada em tempo. Um documento é visto como um conjunto de eventos concorrentes dependentes de tempo, como mídias, por exemplo, conectados por hiperlinks. Seguindo o princípio do SGML, o HyTime é independente de qualquer representação de textos em geral.

Os conceitos de hypermídia representados pelo HyTime incluem **[YN99]**:

- Localização complexa de objetos documentos;
- Relacionamentos (hiperlinks) entre objetos documentos e;
- Objetos numéricos, além de associações de medida entre objetos documentos.

1.2.3 A linguagem HTML

O HTML foi originalmente desenvolvido por Tim Berners-Lee enquanto ele estava no CERN e seu uso foi popularizado pelo navegador Mosaic devolvido no NCSA. Durante os anos 90, seu uso floresceu com o crescimento exponencial da Web. Durante esta época, o HTML foi expandido para versões mais sofisticadas **[W3C-a]**.

O HTML é na verdade uma versão simplificada do SGML. Os documentos HTML são simples arquivos de texto ASCII com etiquetas de apresentação embutida **[HK96]**. Para

também dar suporte a multimídia, foi também incorporado o padrão HyTime. Com isso, o HTML é simples e adequado para hipertexto, multimídia e para exibir documentos em geral. Seus documentos também podem ter mídias agregadas, tais como som e imagens em vários formatos. Também pode ter campos para metadados que podem ser utilizados por diferentes aplicativos e propósitos. Podemos ainda adicionar programas (como Javascript) dentro da página, que alguns autores passaram a denominar de HTML dinâmica (DHTML – *Dynamic* HTML).

Definido em 1992, ele ainda está em evolução. Por exemplo, em 1997 foi introduzido o *Cascade Style Sheets* (CSS) **[W3C-b]**. Deve-se a isso não haver uma representação fixa (padrão) para um estilo de documento. O CSS oferece um meio poderoso e manipulável para que autores, artistas e tipógrafos possam criar efeitos visuais que melhoram a estética das páginas Web. O XML (*eXtensible Markup Language*) é também um subconjunto da SGML. No entanto, diferentemente do HTML, ele não prescreve o conjunto de etiquetas permitidas e este conjunto pode ser especializado quando necessário. Esta é a principal característica na representação e troca de dados no XML, onde o HTML é utilizado primariamente para formatação de documentos. Ele pode representar dados de banco de dados, bem como outros tipos de estruturas de dados utilizados em aplicações de negócios **[SKS02]**.

Aplicações HTML típicas usam como padrão um subconjunto de etiquetas de acordo com a especificação simples do SGML. Isto permite aos usuários se liberar para a especificação da linguagem fora do documento e torna muito fácil de embutir aplicações, mas tem um custo da limitação severa do HTML em alguns aspectos importantes **[YN99]**. Em particular o HTML não é capaz de:

- Permitir ao usuário especificar suas próprias etiquetas ou atributos a fim de parametrizar ou qualificar semanticamente seus dados de outro modo;
- Suportar a especificação de estruturas aninhadas necessárias para representar esquemas de banco de dados ou hierarquias orientadas a objetos;
- Suportar o tipo de especificação de linguagem que permite aplicações que verificam dados para validação estrutural em importação.

1.2.3.1 Edição de documentos HTML

Os documentos em HTML são basicamente arquivos ASCII comuns, que podem ser editados por um editor de texto simples, como o Notepad ou Edit. O documento produzido deve ter extensão de formato de arquivo *html* ou *htm*. Para documentos muito complexos, se exige um conhecimento razoável de HTML, além de criatividade e bastante tempo no desenvolvimento de páginas inteiras. Para facilitar esse tipo de trabalho, existem editores especializados, capazes de orientar o usuário durante a elaboração do código. Já outros, mais complexos, são capazes de gerar as marcações HTML automaticamente. Ao clicarmos em um botão, por exemplo, o programa já insere um marca e posiciona o cursor dentro da marca para que possamos inserir o texto que ficará em negrito. Alguns programas desse tipo são o HotDog **[HD]**, HomeSite **[HS]**, AceHTML **[Ace]**, DominHTML **[Dom]**, 1st Page 2000 **[1st]** entre outros. Existem ainda os editores HTML conhecidos como **WYSIWYG**, iniciais de **What You See Is What You Get** (**O Que Você Vê é o Que Você Tem**). Nesses programas digitamos o conteúdo do documento e inserimos a formatação como em um editor de texto mais elaborado com o WordPad ou o MS Word. Mas estes dois não são muito adequados, pois introduzem caracteres especiais que não são reconhecidos pelo navegador. Existem aplicativos mais específicos que podemos editar e visualizar o código HTML – são o DreamWeaver da Macromedia **[Dre],** Sothink **[Sot]**, Cool Page **[CP]** e o FrontPage da Microsoft **[MSFP]**. Porém, tais programas também podem gerar códigos que não compatíveis com todos os navegadores, além de sobrecarregar o documento com marcações desnecessárias.

1.2.3.2. Componentes de um documento básico HTML

A sintaxe de um documento HTML básico apresenta os seguintes componentes, como mostra a Figura 1.1.

```
<html>
<head><title>Título do Documento</title></head>
<body>
texto
mídias
hiperlinks
scripts
.
.
.
</body>
</html>
```

Figura 1.1 - Sintaxe de um documento básico em HTML

As etiquetas HTML não distinguem entre caracteres maiúsculos e minúsculos ou mesclados. A nova versão do HTML, denominada XHTML, já padronizou todas as palavras reservadas com caracteres minúsculos.

1.2.3.3 O navegador e o HTML

O navegador para executar um documento HTML, deve conter um interpretador HTML para exibí-los. Existem também outros interpretadores que são opcionais. A entrada desse interpretador consiste em um documento que se ajusta à sintaxe HTML; a saída consiste em uma versão formatada do documento em uma interface com o usuário, geralmente uma interface gráfica. O interpretador trata dos detalhes de layout que traduz as especificações HTML em comandos que são apropriados ao hardware de exibição do usuário. Os componentes mais importantes de um navegador da Web são esquematicamente exibidos na Figura 1.2 a seguir **[Com01]**:

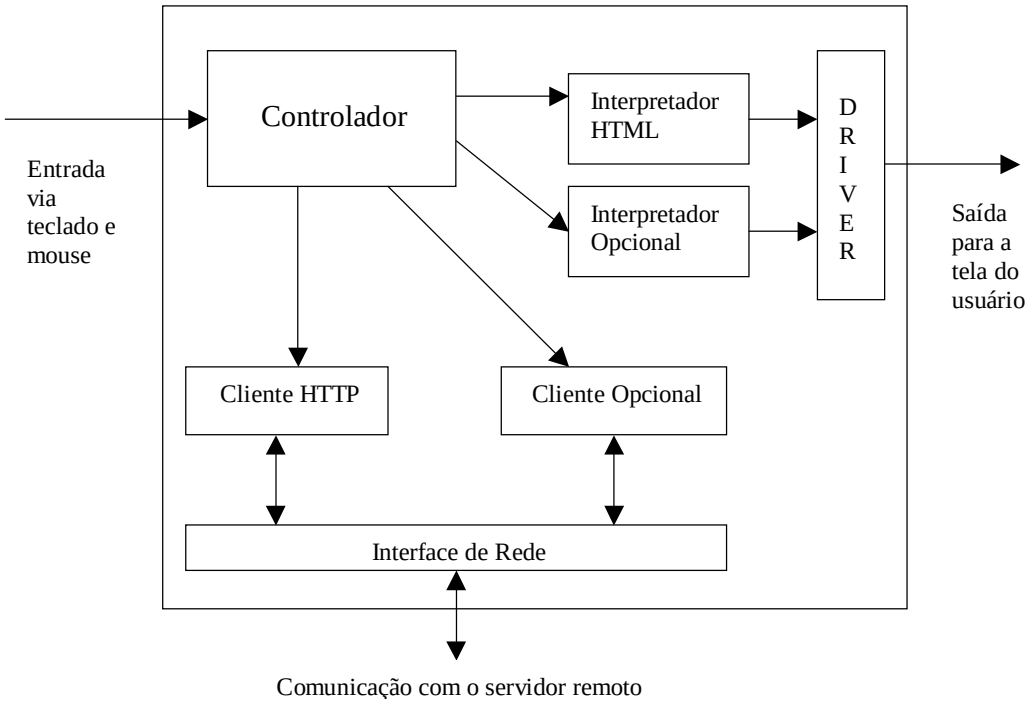


Figura 1.2 - Arquitetura básica de um navegador **[Com01]**

1.3 URL – Uniform Resource Locator

Páginas Web podem conter apontadores para outras páginas da própria Web. Quando a World Wide Web foi criada, ficou aparente de imediato que fazer uma página apontar para outra exigia mecanismos de nomenclatura e localização de páginas. Surgiram questões tipo: Qual o nome da página? Onde a página está reside? Como a página pode ser acessada?

Para resolver o problema da identificação foi criado o **Uniform Resource Locator**. Cada página passa a ter um URL que funciona como um nome universal. Ele é composto por três partes:

- O protocolo;
- O nome do DNS da máquina em que a página está localizada;
- O nome local que indica a página específica, que geralmente é um nome de arquivo na máquina onde ele reside.

Consideraremos o seguinte exemplo:

http://www.exemplo.com.br/figuras/foto.jpg

Este URL é sintaticamente formada pelas três partes: o protocolo HTTP, o nome DNS do *host* *www.exemplo.com.br* e o nome do caminho e do arquivo */figuras/foto.jpg*.

Muitas páginas têm atalhos internos para nomes de arquivos. Nestas, um nome de arquivo nulo representa como padrão a página principal de uma organização. Normalmente, quando o arquivo nomeado é um diretório, isso implica um arquivo denominado *index.hmtl* **[Tan03]**.

1.4 O protocolo HTTP e páginas Web

O HTTP, protocolo de camada de aplicação da Web, está implementado em dois programas: um programa cliente e um programar servidor. Estes dois programas funcionam em diferentes sistemas finais e se comunicam pela troca de mensagens. O HTTP define a estrutura dessas mensagens e o modo como o cliente e o servidor as trocam.

Uma página Web é constituída de objetos. Um objeto é simplesmente um arquivo – tal como um arquivo HTML, uma imagem JPEG, uma imagem GIF, um applet Java, um clipe de áudio e assim por diante – que se pode acessar por meio de um único URL. A maioria das páginas é constituída de um arquivo base HTML e diversos objetos relacionados. Por exemplo, se uma página contiver um texto HTML e cinco imagens JPEG, então esta página tem seis objetos. O arquivo base HTML relaciona os outros objetos da página com seus URLs.

Um navegador é um agente usuário para Web. Ele apresenta a página solicitada e fornece numerosas características de navegação e configuração. Ele implementa o lado cliente do HTTP. Um servidor Web abriga objetos Web, cada um endereçado por um URL. Os servidores também implementam o lado servidor do HTTP.

O HTTP define como os clientes Web solicitam páginas aos servidores e como estes as transferem para os clientes **[KR01]**.

1.4.1 HTTP – Conexões não persistentes

Quando é solicitada uma requisição de página, tem-se um percurso da transferência dela de um servidor para um cliente para o caso de conexões não persistentes.

Vamos enumerar este percurso com o seguinte exemplo utilizando o URL *http://www.algumaInstituicao.com.br/departamento/setor.index*, para adquirir dez objetos JPEG (fotos) que estão em um arquivo base HTML no mesmo servidor, totalizando onze objetos **[KR01]**:

1. O cliente HTTP (o navegador) inicia uma conexão TCP com o servidor *www.algumaInstituicao.com.br* . O número de porta 80 é usado como número de porta *default* na qual o servidor HTTP estará na escuta pelos clientes HTTPs que queiram buscar documentos usando o HTTP;
2. O cliente HTTP envia uma mensagem de requisição HTTP ao servidor através da porta associada à conexão TCP que foi estabelecida no item 1. A mensagem de requisição inclui o nome do caminho */departamento/setor.index*;
3. O servidor HTTP recebe a mensagem de requisição através da porta associada à conexão que foi estabelecida no item 1, recupera o objeto */departamento/setor.index* de seu sistema de armazenamento (memória RAM, disco magnético ou ótico), encapsula o objeto em uma mensagem de resposta HTTP e envia a mensagem de volta ao cliente através da porta;
4. O servidor HTTP ordena que o TCP encerre sua conexão;
5. O cliente HTTP (o navegador) recebe a mensagem de resposta. A conexão TCP é encerrada. A mensagem mostra que o objeto encapsulado é um arquivo HTML. O cliente extrai o arquivo da mensagem de resposta e analisa o arquivo HTML e encontra a referência a dez objetos JPEG;
6. Os primeiros quatro itens são repetidos para cada um dos objetos JPEG referenciados.

À medida que o navegador recebe a página Web, ele a apresenta ao usuário. Os itens apresentados usam conexões não persistentes porque cada conexão TCP é fechada depois que

o servidor envia o objeto e a conexão não persiste para outros objetos. A cada conexão TCP transporta exatamente uma mensagem de requisição e uma mensagem de resposta. Com isso, no caso do exemplo anterior, são geradas onze conexões TCP.

No lado do servidor, as seguintes etapas são executadas:

1. Aceitar uma conexão TCP de um cliente (o navegador);

2. Obter o nome do arquivo solicitado (que está no disco);

3. Retornar o arquivo ao cliente;

4. Encerrar a conexão TCP.

A versão 1.0 do HTTP utiliza a conexão não persistente. Quando a Web consistia em apenas em texto HTML, esse método era adequado. Depois que ela passou a conter mídia com inúmeros ícones, imagens, som e outros itens visuais, este tipo de conexão se tornou extremamente inadequado, pois para cada item era introduzido um atraso devido a várias ativações e desativações de conexão do TCP.

1.4.2 HTTP – Conexões persistentes

Ao contrário das conexões não persistentes, nas conexões persistentes, o servidor deixa a conexão TCP aberta após ter enviado uma resposta. As requisições e respostas subseqüentes entre os mesmos clientes e o servidor podem ser enviadas por meio da mesma conexão. Neste caso, uma página Web inteira (arquivo HTML mais seus objetos) pode ser enviada mediante uma única conexão TCP persistente. Também, múltiplas páginas que residem no mesmo servidor, podem ser enviadas por meio de uma única conexão.

Há dois tipos de conexões persistentes:

- Sem paralelismo;
- Com paralelismo.

No caso da conexão sem paralelismo, o cliente lança nova requisição somente quando a resposta prévia é recebida. Cada um dos objetos relacionados sofre uma RTT (***Round-trip-time*** – tempo de percurso de ida e volta) para solicitar e receber o objeto. Mesmo tendo esse

pequeno atraso, esse tipo de conexão ainda assim sofre menos atrasos que a conexão não persistente.

O modo *default* do HTTP/1.1 usa conexões persistentes com paralelismo. Neste caso o cliente HTTP lança uma requisição assim que ele encontra uma referência. Deste modo, ele pode fazer requisições completas para os objetos relacionados. Quando o servidor recebe as conexões, ele pode enviar todos os objetos. Se todas as requisições forem enviadas completas e todas as respostas forem devolvidas da mesma maneira, então se gastará apenas um RTT para todos os objetos relacionados (em vez de um RTT por objeto relacionado como quando não se está usando paralelismo). Além disso, a conexão TCP com paralelismo fica ativa por uma fração menor de tempo. Além da redução dos atrasos, as conexões persistentes, com ou sem paralelismo, têm um atraso menor de partida lenta do que as conexões não persistentes. A razão para isso é que o servidor persistente, após ter enviado o primeiro objeto, não tem de enviar o próximo objeto à taxa inicial lenta, já que continua usando a mesma conexão. Em vez disso, o servidor pode retornar a mesma taxa com que o primeiro objeto foi enviado **[KR01]**.

1.5 DNS – O serviço de diretório da Internet

Assim como os seres humanos podem ser identificados de muitas maneiras, também podem os hospedeiros da Internet. Um identificador de hospedeiro é um nome de hospedeiro (*hostname*). Para facilitar a localização destes hospedeiros foi criado o DNS – *Domain Name System* (Sistema de Serviço de Domínio). Nome de hospedeiros como *www.cnn.com*, *www.yahoo.com* e *www.msn.com* são fáceis de lembrar e, portanto são mais naturais aos seres humanos. Todavia, nomes de hospedeiros fornecem pouca informação sobre a localização de um hospedeiro na Internet. Além disso, como os nomes de hospedeiro podem consistir em caracteres alfanuméricos de comprimento variável, eles são difíceis de serem processados pelos roteadores. Por essas razões, os hospedeiros também são identificados por seus endereços IP. Um endereço IP é constituído de quatro bytes e tem uma rígida estrutura hierárquica. A forma de um endereço IP tem a forma XXX.XXX.XXX.XXX, onde cada ponto separa um dos bytes em notação decimal cujo valor varia de 0 a 255. Um endereço IP é hierárquico porque, à medida que examinamos o endereço da esquerda para a direita, obtemos gradativamente informações específicas sobre onde o hospedeiro está localizado na Internet.

É semelhante ao endereçamento postal, aonde o código vai de cima para baixo para obtermos a localização mais específica de uma residência **[KR01]**. Segundo **[Com01]**, o número de segmentos em um nome de domínio corresponde à hierarquia de nome. Não existe nenhum padrão universal porque cada organização pode escolher a forma como estruturar nomes em sua hierarquia. Além disso, nomes dentro de uma organização não precisam seguir um padrão uniforme porque grupos individuais dentro da organização podem escolher uma estrutura hierárquica que é a melhor apropriada ao grupo.

1.5.1 Serviços fornecidos pelo DNS

As pessoas preferem trabalhar com o identificador de nome de hospedeiro por ser mais fácil de trabalhar e lembrar, ao passo que os roteadores preferem os endereços IP de comprimento fixo e estruturado hierarquicamente. Para conviver com essas duas formas de endereços IP, é necessário que um serviço diretório que traduza os nomes de hospedeiro para endereços IP. Esta é a tarefa principal do DNS da Internet. O DNS é:

1. Um banco de dados distribuído, implementado em uma hierarquia de serviços de nome;
2. Um protoloco de camada de aplicação que permite que hospedeiros e servidores de nome se comuniquem para poder fornecer o serviço de tradução.

Os servidores de nome são freqüentemente máquinas Unix que rodam o software **Bind** (*Berkeley Internet Name Domain* – Domínio de Nome da Internet de Berkeley). O protocolo DNS utiliza o UDP e usa a porta 53.

O DNS é comumente empregado por outros protocolos de camada de aplicação – inclusive HTTP, o SMTP e o FTP - para traduzir nomes de hospedeiros fornecidos por usuários para endereços IP. Como exemplo, considere-se o que acontece quando um navegador (cliente HTTP), rodando em uma máquina de algum usuário, requisita o URL *www.empresa.com.br/setor/dc/index.html*. Para que a máquina do usuário possa enviar uma mensagem de requisição HTTP ao servidor Web *www.empresa.com.br*, ela precisa obter o endereço IP dele. Isso é feito da seguinte maneira: a própria máquina do usuário roda o lado cliente da aplicação DNS. O navegador extrai o nome de hospedeiro, *www.empresa.com.br*,

do URL e passa o nome para o lado cliente da aplicação DNS. Como parte de uma mensagem de pesquisa, o cliente DNS envia uma consulta com o nome de hospedeiro para um servidor DNS. Ele recebe uma resposta, que inclui o endereço IP referente àquele nome de hospedeiro. O navegador então abre uma conexão TCP para o processo servidor HTTP localizado naquele endereço IP. O DNS adiciona um atraso – às vezes significativo – às aplicações de Internet que o utilizam. Felizmente, o endereço IP procurado está com freqüência na memória cachê de um servidor de nome DNS adjacente, o que ajuda a reduzir o tráfego na rede, bem como o atraso médio provocado pelo próprio DNS **[KR01]**.

1.6 Um breve histórico da Internet e da World Wide Web

Em meados da década de 60, no auge da Guerra Fria, o Departamento de Defesa dos Estados Unidos concebeu uma rede de controle e comando capaz de sobreviver a uma guerra nuclear. As tradicionais redes de telefone de comutação de circuitos eram consideradas muito vulneráveis, pois a perda de uma linha ou comutação provavelmente encerraria todas as conversações que a estivessem utilizando e a rede poderia ser particionada. Para resolver este problema, o Pentágono convocou sua divisão científica, a ARPA (*Advanced Research Projects Agency*) para desenvolver um novo projeto.

A ARPA foi criada depois que a União Soviética lançou o Sputnik, em 1957; sua missão era desenvolver tecnologias que pudessem ser usadas com fins militares. Na verdade a ARPA não tinha cientistas ou laboratórios de pesquisa; tinha apenas um escritório e um pequeno orçamento. Seu trabalho era subvencionar pesquisas nas universidades e empresas cujas idéias lhe parecessem promissoras **[Tan03]**.

Até o início da década de 90, a Internet era praticamente restrita a pesquisadores ligados às Universidades, ao governo e à Indústria. Com o advento do IBM PC, de uma nova aplicação, a World Wide Web – WWW – e do primeiro navegador com interface gráfica – o Mosaic [**Mos**], e posteriormente o Netscape [**Net**] - a Internet passou a crescer exponencialmente e ter milhões de usuários em todo mundo.

A década de 90 estreou com dois eventos que simbolizaram a evolução contínua e a comercialização iminente da Internet. A ARPAnet, a progenitora da Internet, deixou de

existir. A Web passou a servir como plataforma para a habilitação e a disponibilização de centenas de novas aplicações, inclusive negociação de ações em bolsa de valores e serviços bancários on-line, serviços sofisticados de multimídia e serviços de aquisição de informação.

A Web foi inventada no CERN (Centro Europeu para Física Nuclear – European Center for Neclear Physics) por Tim Berners-Lee no período de 1989 a 1991 com base em idéias originadas em trabalhos realizados por Bush **[Bus45]** na década de 40 e por Ted Nelson **[ZD98]** na década de 60 sobre hipertexto. Berners-Lee e seus colegas desenvolveram as versões iniciais de HTML, HTTP, um servidor para a Web e um *browser* (navegador). Os navegadores originais do CERN não tinham interface gráfica, tudo era solicitado no prompt via de linhas de comando. Visando melhorar a interação com os navegadores, vários pesquisadores começaram a desenvolver as primeiras interfaces gráficas GUI (Graphical User Interface - Interface Gráfica de Usuário). Marc Andreessen liderou uma equipe que desenvolveu o popular navegador Mosaic que teve sua primeira versão em 1993. Em 1994, ele e James Baker fundaram a empresa Mosaic Communication, que posteriormente se tornou a Netscape Communications Corporation. Em 1996, a Microsoft introduziu na Web o seu navegador Internet Explorer **[MSIE]**, tirando o domínio da Netscape **[W3C-c]**.

1.7 Busca na Web

Quando a World Wide Web surgiu na década de 80, não se imaginava que ela viesse a se utilizada de forma tal como ela é hoje. Grande quantidade de informação trafega pela grande teia mundial diariamente. Neste contexto, a busca pela informação é um dos aspectos mais importantes, pois pesquisar sobre um determinado assunto num gigantesco mar de documentos eletrônicos não é uma tarefa trivial. A busca pela informação tem como base de partida uma palavra chave ou palavras chaves. Neste contexto, cabe a teoria de re-aquisição de informação (*Information Retrieval*) preocupar-se em fornecer melhores algoritmos de busca.

Basicamente há três formas de busca na Web. Duas delas são bem conhecidas e são freqüentemente utilizadas. A primeira utiliza engenhos de buscas que indexam uma porção de documentos da Web como um banco de dados de texto completo. A segunda utiliza diretórios

que classificam documentos selecionados da Web por assunto de interesse. E a terceira, ainda não totalmente disponível, utiliza estruturas de hiperlinks para realizar suas buscas **[YN99]**.

1.7.1 Busca por meio de hiperlinks

Um paradigma para realizar uma busca na Web é baseado na exploração de hiperlinks. Ele aborda linguagens de consulta estruturada Web Query e a busca dinâmica.

O sistema WebQuery é composto por alguns componentes. Estes componentes são agrupados em dois subsistemas, correspondendo a duas fases do sistema, atividade de pré-processamento e ambiente de execução (*run-time*) como ilustra a Figura 1.3. A tarefa primária da fase de pré-processamento é a coleta de informação de conectividade da World Wide Web, através de um *spider* (rastreador). Esta informação de conectividade especifica simplesmente, para cada nó, o conjunto inteiro de nós que se interligam. Esta informação é então pós-processada para remover as informações redundantes e dados não interessantes (não interessantes do ponto de vista estrutural, tais como vínculos para documentos não HTML). Finalmente, o banco de dados é gerado a partir da conectividade dos dados para re-aquisição imediata da informação na fase de execução. Nessa fase, o mecanismo de busca (*search engine*) é usado para submeter uma palavra-chave (ou conteúdo) para realizar a varredura **[VP]**.

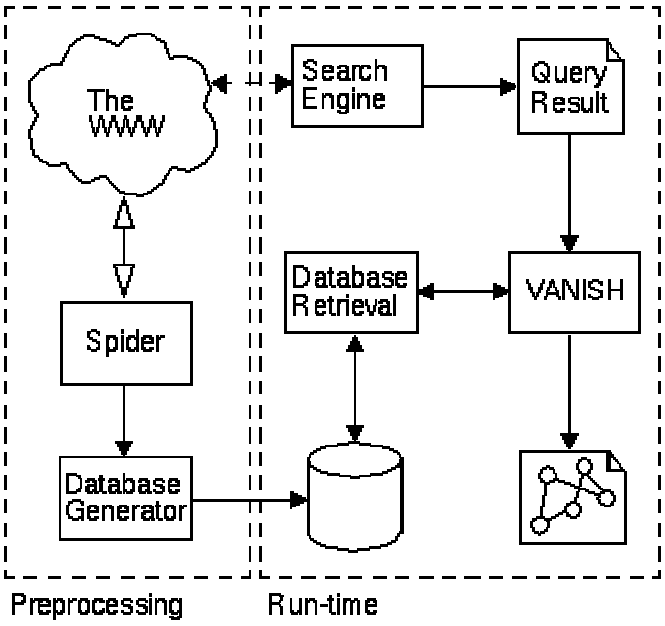


Figura 1.3 – Estrutura do sistema *WebQuery* **[VP]**

1.7.1.1 Linguagens Web Query

A maior parte das consultas está baseada no conteúdo de cada página. Entretanto, consultas também podem incluir estruturas de ligações que conectam páginas Web. Por exemplo, poderíamos querer procurar por todas as páginas Web que contém pelo menos uma imagem e que são alcançáveis a partir de uma dada página que disponibiliza pouco menos de três vínculos. Diferentes modelos de dados têm sido propostos para este tipo de consulta. Os mais importantes são aqueles que utilizam um modelo de grafo rotulado para representar páginas Web (nós) e hiperlinks (arestas) entre estas; e um modelo de dados para representar o conteúdo das páginas. Embora alguns modelos e linguagens para consulta de hipertexto foram propostas antes do aparecimento da Web, a primeira geração dessas linguagens foi direcionada para combinar conteúdo com estrutura. A segunda geração dessas linguagens, denominada linguagens de manipulação de dados Web, mantinham a ênfase em dados semi-estruturados. Entretanto, elas foram estendidas para suportar consultas à estrutura das páginas Web e para permitir a criação de novas estruturas como resultado de uma consulta. São exemplos destas, a STRUQL **[Fer+97]**, a FLORID **[Hin+97]** e a WebOQL **[AM98]**. Todas

essas linguagens não são utilizadas diretamente pelos usuários e sim por meio de aplicativos. Muitas dessas linguagens têm sido ainda estendidas para atender outras atividades na Web, tais como extração e integração de informação de páginas **[YN99]**.

1.7.1.2 Busca dinâmica

Busca dinâmica na Web é equivalente a uma busca seqüencial em texto. A idéia é utilizar um busca online para descobrir informações relevantes por meio de ligações. A principal vantagem é que podemos buscar na estrutura corrente da web o que não está contido nos índices de um engenho de busca. Para realizar estas buscas são utilizados os chamados Agentes de Software (*Software Agents*) que fazem uso de heurísticas em seus algoritmos para realizar uma busca. A idéia básica destes algoritmos é seguir ligações com certa prioridade, partindo de uma simples página e de uma determinada consulta (*query*). A cada passo, a página com maior prioridade é analisada. Se esta não for relevante, a heurística decide se vai ou não seguir as ligações da página. Se decidir seguir, novas páginas são adicionadas na lista de prioridades em posições apropriadas **[YN99]**.

1.8 Sistemas distribuídos e a Web

Como as páginas HTML estão distribuídas pelos diversos servidores na Web, não é demais lembrar algumas características dos sistemas distribuídos. Então, podemos caracterizá-lo conforme as seguintes funcionalidades:

- **Compartilhamento de recursos** (*Resource sharing)*: define o conjunto de objetos que podem ser compartilhados em um ambiente colaborativo. Esses objetos variam de componentes de *hardware* tais como discos rígidos, impressoras; e estruturas de *software* como arquivos, janelas, bases de dados ou outros objetos de informação;
- **Abertura** (*Openness):* essa característica determina a forma como o sistema pode ser estendido. A abertura em sistemas distribuídos é atingida pela especificação e documentação das *interfaces* mais importantes do sistema, tornando-as públicas para os desenvolvedores. Um sistema pode ser considerado aberto ou fechado com

relação à extensão de *hardware*, por exemplo, com a adição de periféricos: memória ou interfaces de comunicação; ou com relação a extensões de software, por exemplo, com a adição de ferramentas nos sistemas operacionais e de protocolos de comunicação.

- **Concorrência** (C*oncurrency):* em sistemas distribuídos, existem vários computadores, cada um possuindo um ou mais processadores. Dessa forma, mais de um processo pode requisitar a utilização de um mesmo recurso, causando o surgimento de concorrência. Nesse contexto, a solução para esse problema se dá através da sincronização do acesso aos recursos compartilhados. Devemos lembrar que o conceito de concorrência não está necessariamente vinculado ao compartilhamento de um único recurso e sim a processamento paralelo.
- **Escalabilidade** (*Scalability):* a necessidade por escalabilidade, ou seja, a capacidade de crescimento, não é simplesmente um problema de desempenho de *hardware* ou rede. Em sistemas de computação centralizados, alguns recursos compartilhados, como memória e processador, são limitados e não podem ser replicados indefinidamente. Em sistemas distribuídos (SD), essa limitação é removida, pois, potencialmente, um sistema distribuído pode possuir um número teoricamente ilimitado de computadores, cada um deles possuindo memória e processador próprios. Entretanto, a elaboração de um SD deve possibilitar uma utilização eficiente de escalabilidade, pois, à medida que o tamanho e a complexidade de um SD aumenta, torna-se difícil mantê-lo eficiente. Tecnologicamente pode haver um limite para o número de máquinas interligadas.
- **Tolerância a falhas**: é a capacidade de um sistema continuar operando quando ocorre alguma falha. As falhas em computadores podem ser de: *hardware* e de *software*. Para desenvolver-se um sistema tolerante a falhas, o projetista deve inserir redundância de *hardware*, substituindo componentes à medida que falhas ocorrem, e projetando *software* com capacidade de recuperação, pois usualmente um sistema encontra-se em situações diferentes do seu funcionamento normal, podendo causar falhas em sua operação.
- **Transparência**: é definida como a capacidade de abstrair do usuário ou de uma aplicação, a localização dos componentes de um sistema distribuído. Esta

característica permite que um sistema distribuído não seja observado como uma coleção de componentes independentes, mas como um sistema único.

1.9 Motivação e objetivos

As páginas Web podem ser vistas como um imenso grafo de nós interconectados através de hiperlinks dinâmicos. Um dos problemas encontrados pelos mecanismos de buscas está relacionado a vincular uma página a outra que não existe mais ou mudou de endereço (seu link foi alterado). Nas especificações HTML do W3C **[W3C-d]** não há nenhuma informação relacionada ao quanto uma determinada página na Web tem hiperlinks apontando para ela. Somente alguns provedores de páginas mantêm uma estatística simples do número de acessos que estes já tiveram desde a sua entrada na Web.

Fazer uma busca de um determinado assunto na Web atualmente é muito dispendiosa e muitas vezes não eficaz. Os servidores de busca implementaram seus mecanismos com algoritmos de rastreamento mais eficientes e com uso de memória cachê com o intuito de reduzir o esforço de pesquisa na Web. No entanto, como os documentos das páginas Web são muito dinâmicos devido a mudanças constantes, o mecanismo com uso dessa memória, apesar de ser eficiente, torna-se ineficaz nessa situação, uma vez que pode não refletir mais os dados atualizados e novos hiperlinks que possivelmente foram incluídos. Em relação aos algoritmos de rastreamento, o maior problema ocorre quando estes seguem hiperlinks de páginas vazias, que foram excluídas pelo suporte do servidor de páginas.

Num servidor de páginas de um portal, por exemplo, a quantidade destas é imensa e o número de atualizações é constante. Páginas Web excluídas em um determinado servidor, pode levar ao que denominamos de “referências pendentes” (“*dangling references*”), onde várias páginas não são mais referenciadas por nenhuma outra página vinculada direta ou transitivamente ao URL. Estas páginas podem ser consideradas lixo, pois não são mais utilizadas e ficam ocupando espaço do diretório do URL desnecessariamente. Para retirar essas páginas desse diretório, o suporte tem que realizar essa atividade praticamente de forma manual, tendo o trabalho árduo de identificá-las previamente, que no caso de um ambiente distribuído é impraticável.

Esta dissertação propõe-se a dar uma solução para esta lacuna dos vínculos dinâmicos entres as páginas Web. Apresenta-se um algoritmo que gerencie a quantidade de ligações que uma determinada página está sendo referenciada em outras páginas. Com isto se teria conhecimento quando uma página fosse retirada do ar ou se tivesse seu hiperlink completamente alterado.

É importante notar que estruturas cíclicas são freqüentemente encontradas entre páginas, pois há uma tendência entre elas de fazerem várias referências entre si. O tratamento adequado de tais estruturas sem que haja uma perda de páginas em memória é fundamental para a exatidão do gerenciamento de páginas Web aqui proposto.

A Figura 1.4 mostra um cenário de páginas Web com hiperlinks entre si, inclusive formando ciclo.

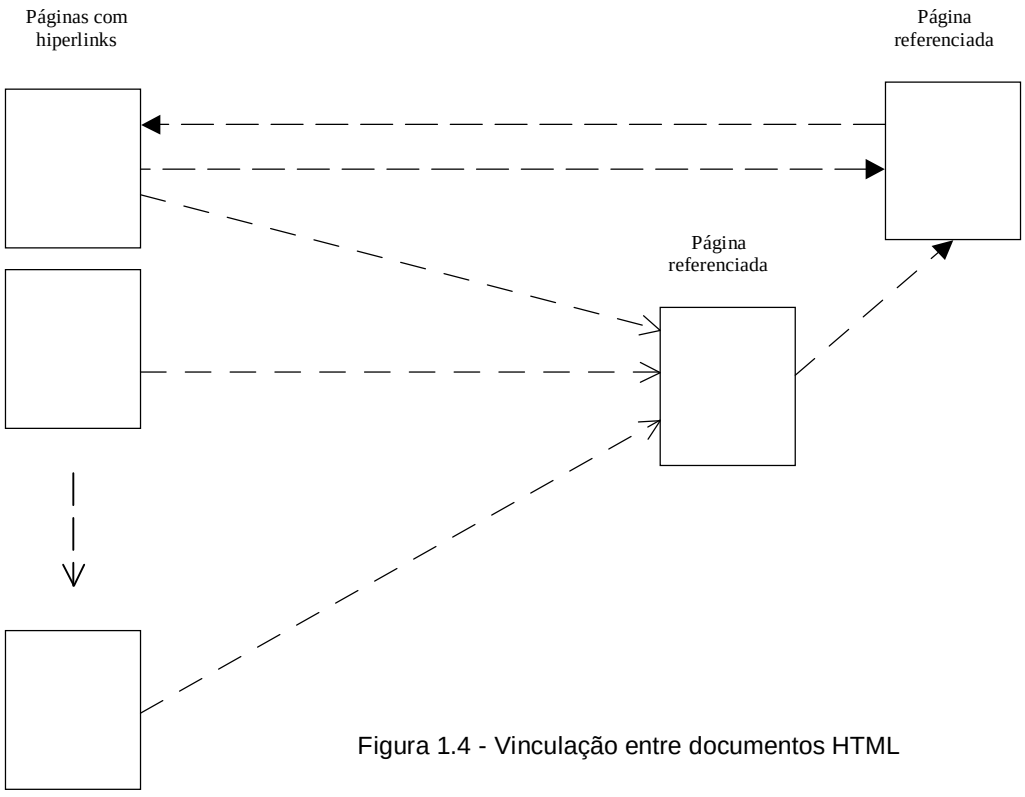


Figura 1.4 - Vinculação entre documentos HTML

1.10 Estrutura desta dissertação

A presente dissertação está dividida em cinco capítulos, sendo o primeiro deles, este capítulo de introdução. Os outros são descritos como se segue:

O capítulo 2 dá uma visão geral dos algoritmos de coleta de lixo para ambiente monoprocessado, iniciando pelos algoritmos mais simples, numa visão seqüencial.

O capítulo 3 aborda os algoritmos para gerenciamento automático de memória em arquiteturas fracamente acopladas tais como redes de computadores.

O capítulo 4 apresenta uma aplicação do algoritmo distribuído de contagem de referências cíclicas apresentado por Lins em **[Lin02]** para o gerenciamento consistente de páginas Web, cujo objetivo é evitar as atuais e freqüentes "*dangling references*". Devido sua real importância nesta dissertação, justifica-se sua descrição neste capítulo e não no anterior.

O capítulo 5 apresenta as conclusões deste trabalho, bem como linhas gerais para trabalhos futuros.

Capítulo 2

Introdução ao gerenciamento automático de memória

O gerenciamento automático dinâmico de memória, também conhecido como *Garbage Collection*, é aquele que é realizado pelo ambiente em tempo de execução (*run-time*).

A gerência automática de memória possui longa e respeitável história desde que surgiu a linguagem LISP **[LIS58]**, viabilizando o uso de estruturas de dados, como listas, que não observam a disciplina de alocação e liberação em pilhas. De alguma maneira, o "estado-da-computação" é analisado e decidida a necessidade das estruturas de dados alocadas. Por exemplo, no caso da linguagem Java **[Jav95]**, a estrutura de um objeto, além dos atributos declarados na classe, possui um cabeçalho (*header*) com atributos especiais usadas somente pela máquina virtual que não estão acessíveis ao desenvolvedor. Como a especificação da JVM (*Java Virtual Machine*) não define um padrão único, cada implementação de uma máquina virtual tem geralmente um ponteiro para a classe do objeto **[LY99]**.

Vale ressaltar que a coleta de lixo não é exclusividade das linguagens funcionais ou de Java; ela envolve certos conceitos e algoritmos que pretendemos explicitar ao longo dessa dissertação. Esse mecanismo era discriminado de certa forma pelos desenvolvedores, pois não se acreditava na sua eficiência e segurança em relação ao gerenciamento "manual" de memória que era realizado através de comandos de desalocação explícita disponível no ambiente da linguagem. Acreditava-se sim, que ele era um consumidor desnecessário de recursos, causando atrasos no processamento. Esse mecanismo só era conhecido pelo uso nas linguagens funcionais e declarativas, cuja preocupação na resolução dos problemas estava mais voltada para a abstração da modelagem e não para as idiossincrasias internas de gerenciamento de memória. Como estas linguagens não eram eficientes em termos de velocidade de processamento, criou-se o mito de que a coleta automática de lixo era um empecilho ao desempenho computacional de processamento. Ao contrário, alguns experimentos foram realizados e publicados (**[App87]**, **[Zor93]**, **[Sou00]**), provando que o

custo benefício entre a abstração da resolução de problemas e a preocupação com detalhes de programação com ponteiros não era de todo inviável.

Rovner **[Rov85]** em seus experimentos mostrou que uma proporção considerável do tempo de desenvolvimento de software é gasto em *bugs* referentes a problema de gerenciamento de memória. Adita-se ainda, o crescimento vertiginoso da velocidade dos processadores nas últimas décadas a favor do gerenciamento automático. A própria comunidade observou que o uso explícito de ponteiros em sistemas grandes e complexos, veio a se tornar uma grande "dor de cabeça" quando ocorrem problemas de falta de memória ou ponteiros pendentes, além de dificultar muito a manutenção dos códigos fontes. No caso dos ponteiros pendentes é um tipo de erro difícil de encontrar e corrigir, uma vez que apresenta os seguintes problemas segundo **[GB+01]**:

- O ponteiro pendente não pode ser desvinculado até bem depois da memória ter sido liberada erroneamente;
- Depois de desvincular o ponteiro pendente, o programa pode prosseguir com os dados incorretos por algum tempo, antes de ser observada uma inconsistência;
- Pode ser difícil reproduzir o erro, porque talvez seja necessária a mesma seqüência exata de alocações e desalocações de memória para provocar o mesmo erro observado.

Além de facilitar o trabalho do programador, a coleta de lixo tem algumas vantagens importantes. Jones e Lins **[JL96]** dão três razões principais a favor da coleta de lixo:

- **Requisitos da linguagem**. Linguagens funcionais, como LISP e Haskell **[Has]**, tem seqüências de execução não previsíveis e a desalocação explícita é geralmente impossível; Neste caso a coleta de lixo é obrigatória;
- **Requisitos do problema**. Um exemplo dado por **[BC92]** ilustra bem a seguinte situação: suponha um tipo geral de pilha de dados a ser implementado em C como uma lista ligada. Se o dado pode ser empilhado em duas pilhas, como poderá o comando *Pop* (desempilha) se comportar em termos de desalocação? Desde que seja possível empilhar ponteiros para o mesmo tipo de dados em ambas as pilhas, não podemos simplesmente liberar a memória toda vez que um Pop é executado. Alguma convenção é requisitada para desalocação mesmo para tal abstração

simples. Isto tanto complicará a interface da pilha, como reduzirá sua aplicabilidade ou forçará cópia desnecessária;

- **Questões de Engenharia de Software**. Um requisito básico para um "bom" software é o encapsulamento. Aplicações orientadas a objetos complexas consistem de componentes que se comunicam através de interfaces claramente definidas. Gerenciamento de armazenamento controlado pelo programador inibe esta modularidade e as linguagens OO mais modernas (como Java) são assistidas por um coletor de lixo. Coletores de lixo têm sido também escritos e disponíveis em bibliotecas para linguagens não cooperativas (sem *multithread*) como C **[C72]** e C++ **[Str91]**.

2.1 O motivo da coleta automática de lixo

As primeiras linguagens de programação de alto nível, tais como Fortran **[FOR57]** e Cobol **[COB60]** em suas versões originais, utilizavam o que foi denominado de alocação estática (*static allocation*) de memória, ou seja, todos os seus procedimentos, variáveis e constantes ocupam porções de memória de modo fixo previamente alocadas durante a fase de compilação. Recursividade e procedimentos dinâmicos inexistem. Posteriormente, surgiu a primeira linguagem estruturada em bloco que utiliza alocação de pilha (*stack allocation*) de memória, o Algol-58 **[ALG58]**. Esse tipo de alocação é uma melhoria sobre a alocação estática, pois alocação é armazenada em pilha, permitindo o uso de registros de ativação através de chamadas de procedimentos, proporcionando sua desalocação após o uso e retornando ao ponto de chamada. Diferentemente da disciplina em que o primeiro que entra é o primeiro que sai em uma pilha, estruturas de dados em uma *heap* podem ser alocadas e desalocadas em qualquer ordem (*heap allocation*). Assim, registros de ativação e estruturas de dados dinâmicas podem ser ativados pelo procedimento que as criou. Atualmente, muitas linguagens de alto nível utilizam tanto a alocação em pilha quanto na *heap*.

O gerenciamento da manipulação das estruturas de dados dinâmicas pode ser feito de forma explícita pelo programador através de comandos disponíveis da linguagem de programação. Em programas grandes e complexos, esta forma de gerenciamento não é satisfatória, pois o programador pode desalocar indevidamente uma área de memória usada

pelo sistema operacional ou gerar referências pendentes (*dangling references*). Lixo pendente na memória pode causar problemas à medida que o aplicativo vai sendo executado por um período mais longo levando ao problema de falta de espaço em memória (*space-leak*). Neste contexto, a gerência automática de memória se torna a alternativa mais segura e satisfatória, liberando o programador para outras atividades mais relevantes vinculadas ao problema que está sendo modelado.

2.2 A terminologia em coleta de lixo

Os valores que um programa pode manipular diretamente são aqueles vinculados aos registradores do processador, aos da pilha do programa e aos vinculados a variáveis globais. Tais localizações contêm referências aos dados da *heap* na forma de **raízes** (***roots***) da computação.

Um pedaço de dado alocado individualmente na *heap* é chamado **objeto, célula** ou **nó**. Um objeto localizado na *heap* é denominado **vivo** (ou alcançável) se seu endereço está vinculado a uma raiz ou existe um ponteiro que o vincula a outro objeto vivo dentro da *heap*.

Objetos que se tornam inalcançáveis durante a execução são denominados **lixo** (***garbage***). O conjunto de células que estão disponíveis para uso é denominado **células livres** (***free-list***).

Usualmente existe a necessidade de se fazer distinção entre o coletor de lixo e parte do programa que faz o trabalho útil. Usaremos o termo **mutador** para o programa do usuário (seguindo a terminologia de Dijkstra **[DL+78]**), desde que – assim como o coletor - seu trabalho seja mudar referências (ou causar mutação) entre objetos.

Para ambientes distribuídos, são usados os termos **espaço, nó** ou ***host*** para referir cada aplicação individual que executa em um espaço de memória separada e é usualmente capaz de executar seu próprio coletor de lixo local. Um objeto (criado numa máquina chamada de proprietário [***owner***]) é acessível em um sistema distribuído via um ***exit item, surrogate*** ou ***stub*** (sobre um nó cliente) e via um ***entry item*** ou ***skeleton*** (no proprietário). Quando falamos de *stubs/skeletons* em RPC, CORBA, RMI etc, estamos primeiramente interessados nas requisições remotas *marshalling-unmarshalling* (emissão com empacotamento de dados -

recepção com desempacotamento de dados). No caso da coleta de lixo distribuída, os *stub-skeletons* são interessantes em diferentes ponto de vista – eles podem ser usados para armazenamento e/ou para propagação de informação adicional sobre referências inter-objetos.

Um **ciclo** consiste num grupo de objetos que fazem referência a objetos do mesmo grupo, mas nem um deles é acessível pela aplicação (todos eles são lixo). Se o grupo se espalha sobre múltiplos espaços, então os trataremos como um **ciclo distribuído**.

Um coletor de lixo **conservativo** é um dos que não se esforça de imediato para reclamar um objeto não alcançável. Um coletor de lixo **incremental** é aquele que se intercala com o processamento do ambiente *run-time*.

2.3 Questões em coleta de lixo

São algumas das propriedades desejáveis em um coletor de lixo ideal:

- **Completude** – todos os objetos (idealmente incluindo componentes de ciclos) que são lixo no início do ciclo de coleta devem ser reclamados por ele no final.
- **Concorrência** – tanto o mutador quanto o coletor não devem ser suspensos; processos distintos de coleta distribuída devem rodar concorrentemente.
- **Eficiência** – custo de tempo e espaço devem ser mínimos.

De acordo com a técnica usada, algumas dessas características serão muito difíceis (ou mesmo impossíveis) de realizar.

Dependendo do grau de interação entre o mutador e o coletor, podemos dividir as técnicas de coleta de lixo em três categorias:

1. **Seqüencial** – algoritmos *stop-and-collect* (pare e colete). O mutador pára a fim de que o coletor entre em ação;
2. **Incremental** – coletas incrementais não suspendem o mutador enquanto se completa a coleta de lixo. Entretanto, um coletor incremental causará pausas no mutador (mas somente por um breve período de tempo) a cada passo do algoritmo

de coleta. Um coletor incremental pode ser chamado de coletor de tempo real se, no pior caso, os tempos de pausa são limitados por constantes específicas do problema;

3. **Concorrente** – esses coletores tem sido desenvolvidos com a finalidade de executar em arquiteturas com múltiplos processadores. O mutador e o coletor são executados por processos separados (mas que ainda precisam sincronizar suas ações).

Quanto à exatidão dos algoritmos, há duas propriedades principais que os certificam:

1. ***Liveness*** (garantia) – todos os objetos que são lixos serão eventualmente coletados;
2. ***Safety*** (segurança) – objetos sem referência serão coletados.

2.4 Algoritmos em ambiente monoprocessado

Esta seção apresenta alguns algoritmos de coleta de lixo para ambiente não distribuído (**[JL96]**, **[Wil94]**). Estes algoritmos foram especificamente projetados para trabalhar em ambiente monoprocessado sem compartilhamento de memória.

Existem vários algoritmos que implementam a coleta de lixo. Alguns deles são considerados clássicos, pois são mais conhecidos e utilizados. Dentre estes estão os de marcação e varredura (*mark-sweep*), o de contagem de referências (*reference count*) e o de cópia de dois espaços (*copying*). O algoritmo de contagem de referências será descrito com mais detalhes por ser fundamental na compreensão do algoritmo que será utilizado no gerenciamento das páginas Web.

2.4.1 Mark-sweep (Marcação-varredura)

O primeiro algoritmo de reclamação automática de lixo foi o *mark-sweep* ou *mark-scan* desenvolvido por John McCarthy **[McC60]**, que utilizava a técnica de tracejamento, para uso na linguagem funcional LISP. Na coleta por traço, são realizadas duas fases. A primeira é a de

marcação, onde as células vivas são percorridas (tracejadas) e marcadas. Na segunda, a *heap* é percorrida em toda sua extensão e a identificação das células não marcadas é realizada:

1. **Fase *Mark*** – Faz a distinção entre objetos vivos e os objetos lixo. Isto é feito por tracejamento – iniciando na raiz e caminhando no grafo de relacionamento de ponteiros - geralmente tanto por caminhamento em **profundidade** (*depth-first*) quanto por caminhamento em **amplitude** (*breadth-first*). Os objetos encontrados são marcados de algum modo: ou alterando bits dentro dos objetos ou pelo armazenamento destes bits num *bitmat* ou outra estrutura de dados;
2. **Fase *Sweep*** – Faz a varredura e reclama (recolhe) o lixo. Uma vez que os objetos vivos tenham sido distinguidos dos objetos que são lixo, a memória é substituída, ou seja, é exaustivamente examinada, para encontrar todos os objetos não marcados (lixo) e reclamar seu espaço.

A Figura 2.1 ilustra o esquema deste algoritmo.

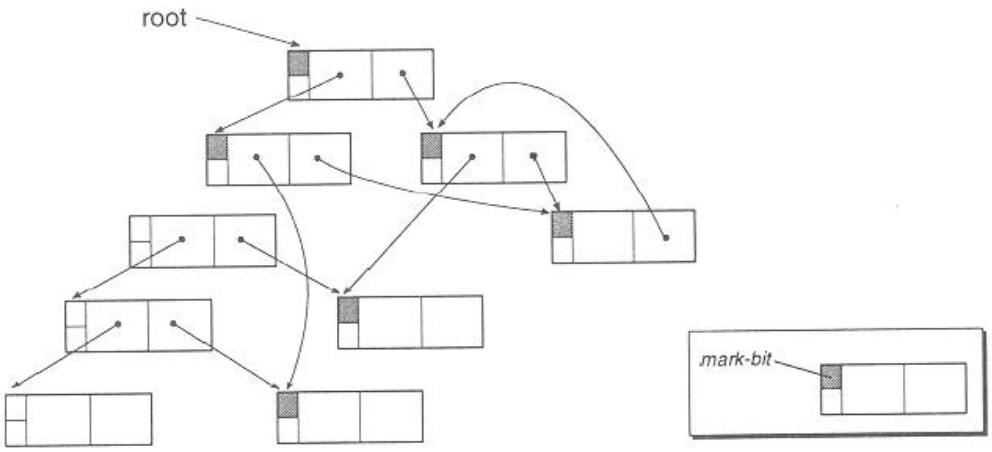


Figura 2.1 – O grafo depois da fase marking. Todas as células *unmarked* (com *mark-bits* não sombreados) são lixo **[JL96]**

Um problema típico com coleta *mark-sweep* é que o custo da coleta é proporcional ao tamanho da *heap*, incluindo tanto objetos vivos e como o lixo. Uma limitação fundamental é

imposta sobre qualquer possibilidade de melhora na eficiência. Uma solução seria usar ou *barreiras de leitura* (que priva o mutador de enxergar um objeto não marcado) ou *barreiras de escrita* (todos os casos perigosos são guardados de modo que o coletor possa (re) visitar os nós em questão).

2.4.1.1 Algoritmo *mark-sweep* - procedimentos

Neste algoritmo, há três atualizações no grafo: A criação, através do procedimento *New*; a marcação e varredura, através do procedimento *Mark-sweep*; e a exclusão, através do procedimento *Delete*.

New(R): Aloca a nova célula R e a vincula ao grafo. O *Mark-sweep* é a rotina de coleta de lixo. O *Allocate*(R) é uma rotina de baixo nível que obtém uma nova célula do *pool* (conjunto) de células livres e faz R apontar para ela.

```
New(R)=
   if (free-pool is empty) then
       Mark-sweep()
   Allocate(R)
```

Algoritmo 2.1 Alocação de uma célula no *mark-sweep*

Mark-sweep(): Inicia a fase de marcação do algoritmo, identificando todas as células ativas através do procedimento recursivo ***Mark***. O ***Sweep*** inicia a fase de busca e retorna células lixo para a *free-list*.

```
Mark-sweep()
  for R in Roots do
     Mark(R)
  Sweep()
  if (free-pool is empty) then
     Abort "memory exhausted"
```

Algoritmo 2.2 O coletor de lixo no *mark-sweep*

Mark (N): Um bit por célula é reservado para uso do coletor, o *mark-bit*. Ele é utilizado para registrar a acessibilidade ou não da célula, a partir da raiz. *Sons*(N) é o conjunto de todas as células M, tal que existe um ponteiro <N,M> que a partir da célula N, se vincula a M.

```
Mark(N) =
   if (Mark-bit(N) == unmarked) then
      Mark-bit(N) =  marked
   for M in Sons(N)
      Mark(M)
```

Algoritmo 2.3 Marcação recursiva simples no *mark-sweep*

Sweep(): Percorre a *heap* linearmente com um todo, de modo que as células não marcadas retornem à *free-list* através da chamada à rotina *free()*. Também desmarca os bits das células ativas.

```
Sweep() =
   for each N in heap do
      if (Mark-bit(N) == marked) then
         Mark-bit(N) = unmarked
      else
         free(N)
```

Algoritmo 2.4 A varredura da *heap* pelo *Sweep*

2.4.2 Contagem de referências

O primeiro algoritmo baseado na contagem de referência foi implementada por Collins **[Col60]** para evitar as paradas demoradas no processo do usuário causada pelo então algoritmo *mark-scan* utilizado em LISP. É um método direto de coleta de lixo incremental que distribuída a carga do gerenciamento de memória durante todo o processamento **[JL96]**.

No algoritmo de contagem de referências cada célula mantém um contador que representa o número de células que a referenciam. Cada vez que uma referência a uma célula é criada, o contador desta é incrementado de uma unidade, e cada vez que esta referência é

excluída, o contador é diminuído de uma unidade. Quando o contador chega a zero, a célula pode ser seguramente reclamada como lixo.

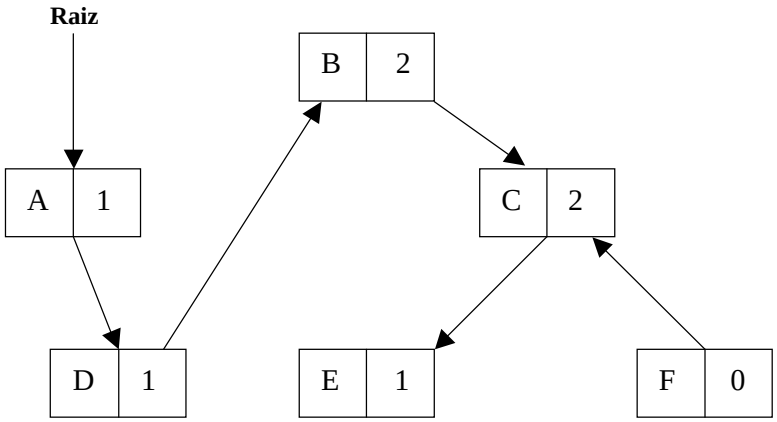


Figura 2.2 – Grafo com células com contadores de referência

O exemplo da Figura 2.2 exibe um identificador e um contador de referências para cada célula. A células C tem seu contador igual a 2 porque duas outras células a referenciam, enquanto o contador de F é 0 (zero), e portanto, esta célula pode ser coletada como lixo.

Esta abordagem é naturalmente incremental para todas as operações, exceto para a exclusão do último ponteiro de uma célula.

A principal desvantagem das abordagens baseadas na contagem de referência é que elas não podem reclamar ciclos – observe que no exemplo apresentado na Figura 2.3, se a referência da célula D para a célula B for excluída, esta ainda teria seu contador igual a 1. Após a esta exclusão, as demais células ainda manteriam seus contadores não zerados e, portanto, elas não serão coletadas como lixo, além de se tornarem inalcançáveis. O método da contagem de referência foi analisado por McBeth **[McB63]** que o apontou como inadequado para tratar de estruturas cíclicas[2].

[2] As estruturas cíclicas surgem do fechamento do grafo em funções recursivas. Embora LISP possua recursividade definida na linguagem, ela era implementada através da “chamada” ao código a cada ponto da recursão via o “script” de forma a evitar ciclos.

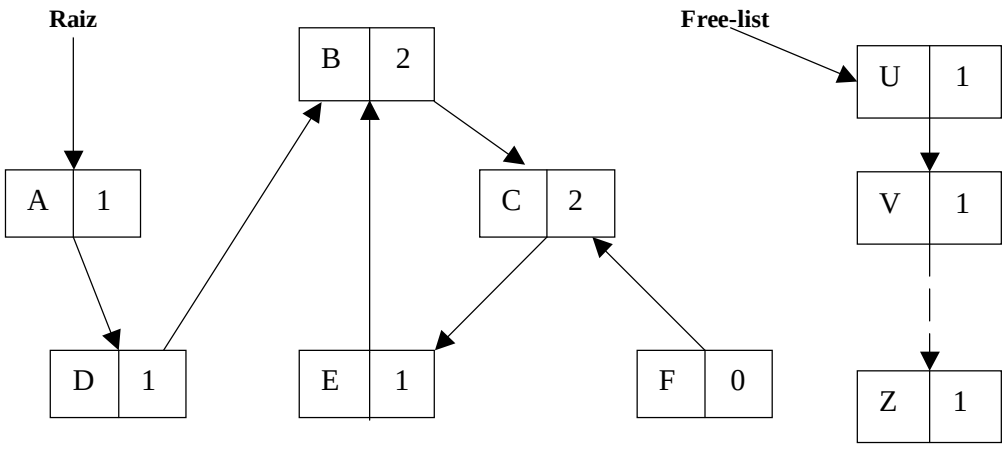


Figura 2.3 – Grafo com células com contadores de referência
com um clico em B, C e E

2.4.2.1 Algoritmo de contagem de referências - procedimentos

Neste algoritmo, para toda célula S, RC(S) é o número de referências a S. A notação <R,S> é usada para denotar um apontador ou ponteiro da célula R para a célula S. Uma célula B está conectada a uma célula A (representada por A **à** B) se, e somente se, existe um ponteiro <A,B>. Uma célula B está transitivamente conectada a uma célula A (representada por A **à** B), se e somente se, existe pelo menos uma célula C tal que (C **à** B) e (A **à** C) ou (A **à** C) é verificado. O ponto inicial do grafo onde todas as células ativas (em uso) estão conectadas é chamada **raiz** (***root***). Inicialmente, todas as células são livres e estarão disponíveis para a alocação com o RC ajustado para um (RC = 1), além de estarem dispostas em uma lista de blocos livres, a ***free-list***. A *free-list* é implementada como uma seqüência de células ligadas entre si por meio de seus apontadores.

Há três atualizações no grafo: A criação, através do procedimento *New*; a cópia, através do procedimento *Copy*; e a exclusão, através do procedimento *Delete*.

New(R): Retira uma célula U da *free-list* e cria o ponteiro <R,U>, onde R é a célula direta ou transitivamente vinculada à raiz. O RC permanece inalterado para uma célula obtida da *free-list* (RC = 1).

```
New(R)=
   Select U from free-list
   Make-pointer <R,U>
```

Algoritmo 2.5 O procedimento *New* do algoritmo de contagem de referência

Copy(R, <S,T>): Cria um ponteiro <R,T>, onde R e S são células transitivamente vinculadas à raiz e <S,T> é um ponteiro do grafo.

```
Copy(R, <S,T>) =
   Make-pointer <R,T>
   Increment RC(T)
```

Algoritmo 2.6 O procedimento *Copy* do algoritmo de contagem de referência

Delete(<R,S>): Remove o ponteiro <R,S> do grafo e realiza o reajuste. Uma célula T pertence ao conjunto definido em Sons(S) – filhos de S se, e somente se, há um ponteiro <S,T>. Quando a última referência a uma célula S é removida, o *Delete* é novamente chamado recursivamente em Sons(S). A célula removida é devolvida à *free-list*.

```
Delete(<R,S>) =
   Remove <R,S>
   Decrement RC(S)
   if (RC(S) == 0) then
      for T in Sons(S) do
         Delete(S,T)
   Link-to-free-list(S)
```

Algoritmo 2.7 O procedimento *Delete* do algoritmo de contagem de referência

2.4.3 Contagem de referências cíclicas

A primeira solução geral dada ao problema da coleta de lixo cíclico foi proposta por Martinez, Wachenchauzer e Lins **[MWL90]**. Algoritmos de contagem de referências cíclicas mais eficientes foram posteriormente propostos por Lins (**[Lin92]**, **[Lin02a]**). Detalharemos o algoritmo de contagem de referências cíclicas nos subitens seguintes, incluindo exemplos dos procedimentos executados.

O Algoritmo de Martinez, Wachenchauzer e Lins **[MWL90]** utiliza a estratégia de realizar uma varredura local no sub-grafo em busca de ciclos através da operação *mark-scan*. Este algoritmo é divido em três fases. Na primeira fase, o sub-grafo sob o ponteiro excluído é percorrido, os contadores são diminuídos devido a referências internas, e a marcação das células já visitadas é completada. Na segunda, o sub-grafo é novamente percorrido e as células com contadores de referências igual a zero são marcadas como possível lixo. As células que ficaram com contador de referências maior que zero possuem referências externas ao sub-grafo. Todas as células que estão vinculadas direta ou transitivamente por referências externas serão marcadas como ativas e terão seus contadores reajustados. E na terceira fase, as células que permaneceram marcadas como possível lixo serão retornadas para a *free-list*. Para realizar estas fases, faz-se uso de um atributo de controle que utiliza três cores. Cada célula passa a ter, além do contador de referência (RC), uma cor (*colour*). Para uma célula S, estes campos são denotados por **RC(S)** e **colour(S)**. As cores são utilizadas para indicar o estado da célula no *mark-scan*, conforme ilustra a Figura 2.3:

Cor	Significado
Verde (*green*)	Em uso (ativa) ou livre
Vermelho (*red*)	Já visitada
Azul (*blue*)	Possível lixo cíclico

Figura 2.4 – Tabela Cor-Significado da contagem de referências cíclicas

2.4.3.1 Algoritmo de contagem de referências cíclicas com *mark-scan* local - procedimentos

Como no algoritmo acíclico, no início todas as células estão na *free-list* já marcadas com *green*.

New(R): Permanece igual ao Algoritmo 2.5 do acíclico.

Copy(R, <S,T>): Permanece igual ao Algoritmo 2.6 do acíclico.

Delete(<R,S>): Quando o contador de referências chega a um, significa que a última referência a célula não compartilhada foi removida. Neste caso, segue-se o mesmo procedimento do algoritmo acíclico. Antes de vincular a célula à *free-list*, o sub-grafo abaixo dela é percorrido recursivamente, para que cada célula que foi referenciada por ela tenha seu contador reajustado. Se ela era uma célula compartilhada, então se executa um *mark-scan* local através das chamadas aos procedimentos *Mark-red*, *Scan* e *Collect-blue*.

```
Delete(<R,S>) =
   Remove <R,S>
   if (RC(S) == 1) then
      for T in Sons(S) do
         Delete(S,T)
      Link-to-free-list(S)
  else
     Decrement RC(S)
     Mark-red(S)
     Scan(S)
     Collect-blue(S)
```

Algoritmo 2.8 O procedimento *Delete* do algoritmo de contagem de referências cíclicas

Mark-red(S): Realiza a primeira varredura no sub-grafo abaixo da célula compartilhada S, pintando as células de vermelho (*red*). Esta cor indica que a célula já foi visitada e está no sub-grafo analisado, podendo futuramente ser identificada como lixo cíclico. Os contadores de referências das células visitadas sofrem decremento e uma nova chamada recursiva é realizada para analisar as células verdes (*green*).

```
Mark-red(S) =
   if (colour(S) == green) then
      colour(S) = red
      for T in Sons(S) do
         Decrement RC(T)
         if (colour(T) == green) then
            Mark-red(T)
```

Algoritmo 2.9 O procedimento *Mark-red* do algoritmo de contagem de referências cíclicas

Scan(S): Percorre o sub-grafo abaixo de S verificando o contador de referências das células visitadas. As células que não tem referências externas são pintadas de azul (*blue*) para indicar possível lixo. Se alguma célula com contador maior que zero for encontrada, ela possui referência externa e o *Scan-green* terá que ser invocado para reajustar todos os contadores das células atingidas e repintá-las de verde.

```
Scan(S) =
   if (colour(S) == red) then
   if (RC(S) > 0) then
      Scan-green(S)
   else
      colour(S) = blue
      for T in Sons(S) do
         if (colour(T) == red) then
         Scan(T)
```

Algoritmo 2.10 O procedimento *Scan* do algoritmo de contagem de referências cíclicas acessa os pontos críticos do grafo diretamente

Scan-green(S): Restaura os valores dos contadores de referências das células pintando-as de verde.

```
Scan-green(S) =
   colour(S) = green
   for T in Sons(S) do
       Increment RC(T)
       if (colour(T) != green) then
          Scan-green(T)
```

Algoritmo 2.11 O procedimento *Scan-green* do algoritmo de contagem de referências cíclicas repinta as células de verde

Collect-blue(S): Recolhe as células para a *free-list* e restaura a cor verde e o contador de referências é reajustado para um.

```
Collect-blue(S) =
   if (colour(S) == blue) then
     colour(S) = green
     RC(S) = 1
     for T in Sons(S) do
        if (colour(T) == blue) then
           Collect-blue(T)
        Remove(<S,T>)
    Link-to-free-list(S)
```

Algoritmo 2.12 O procedimento *Collect-blue* do algoritmo de contagem de referências cíclicas recolhe as células à *free-list*

O maior problema deste algoritmo é que age de forma estrita, ou seja, sempre percorre o sub-grafo toda vez que uma referência a uma célula compartilhada é excluída. Este procedimento torna o processamento do algoritmo muito caro e inviável **[JL96]**.

A quantidade de varreduras que vão ocorrem no sub-grafo após a exclusão de uma célula compartilhada varia de duas no melhor caso, a três no pior. O melhor caso ocorrerá quando o procedimento *scan* encontrar um caso de uma célula com contador maior que um logo depois do *Mark-red*, quando o procedimento *Scan-green* é invocado de imediato para ajustar os contadores e as cores das células. Como não haverá células azuis, também não será invocado o procedimento *Collect-blue*. O pior caso ocorrerá quando logo após o *Scan*, logo depois das células terem sido pintadas de azul, onde haverá uma célula com uma referência externa que vai invocar o procedimento *Scan-green* que vai desfazer todo o trabalho de marcação repintando as células de verde.

Analisaremos através de alguns exemplos a seguir, os cenários de funcionamento do algoritmo de contagem de referências cíclicas, no qual veremos os casos com ciclos coletados e mantidos.

2.4.3.2 Algoritmo estrito - Cenário do melhor caso: ciclo coletado

A Figura 2.5 ilustra o grafo inicial em que vai ocorrer o melhor caso.

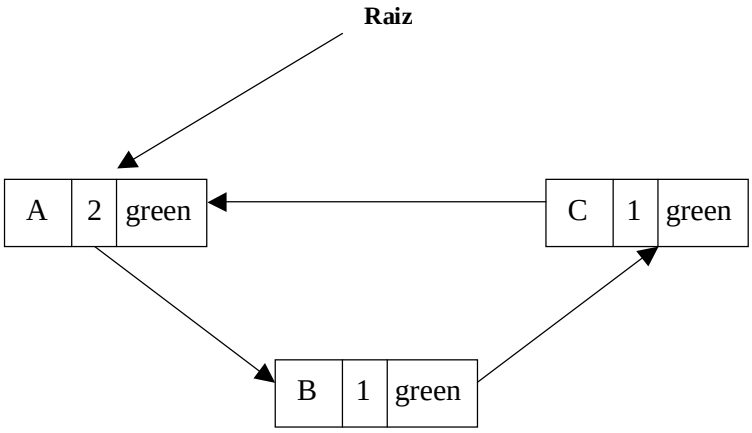


Figura 2.5 – Grafo em que a será feita coleta total

Com a exclusão do ponteiro que vincula a célula A à raiz seja removido. Com a célula A é compartilhada (tem o contador maior que 1), a marcação começa com a invocação recursiva do procedimento *Mark-red(A)* que vai configurar o grafo ilustrado na Figura 2.6, deixando as células pintadas de vermelho e os contadores atualizados.

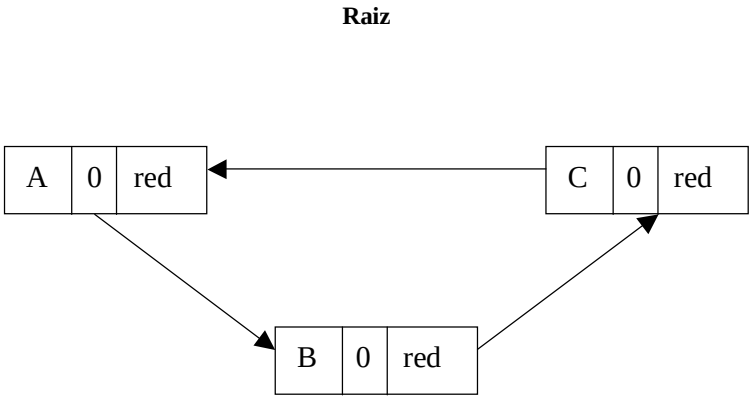


Figura 2.6 – Grafo após a invocação do procedimento *Mark-red*

Logo em seguida, o procedimento *Scan* é chamado para pintar as células de azul, ficando o cenário conforme ilustra a Figura 2.7. Depois, é invocado o procedimento *Collect blue* que fará a coleta das células à *free-list*.

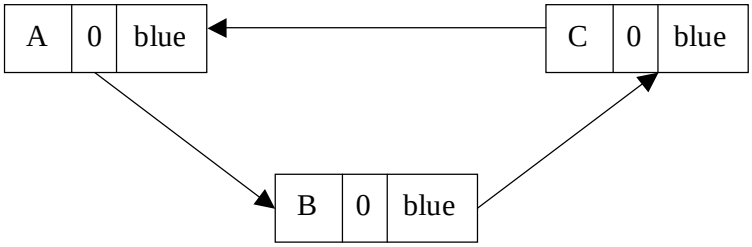


Figura 2.7 – Grafo após a invocação do procedimento *Scan*

2.4.3.3 Algoritmo estrito - Cenário do pior caso: ciclo mantido

Agora analisaremos o cenário em que ocorre o pior caso, que é ilustrado a partir da Figura 2.8, onde a célula C também está vinculada à raiz.

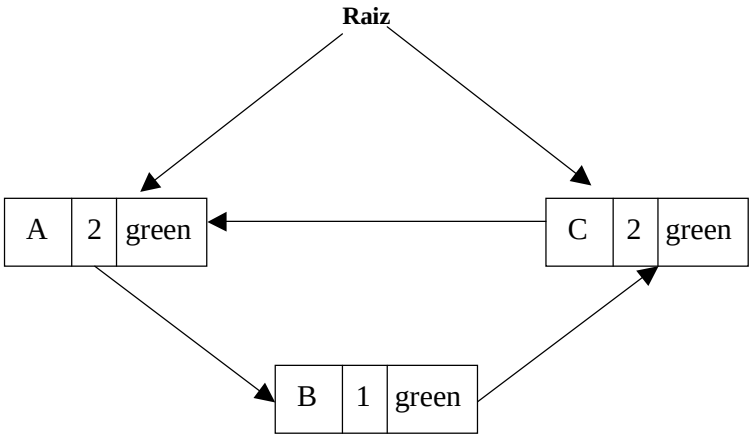


Figura 2.8 – Grafo em que não será feita a coleta total

Neste cenário, repete-se a remoção do ponteiro que vincula a célula A à raiz. O procedimento *Mark-red(A)* é invocado recursivamente logo em seguida, deixando o grafo conforme ilustrado na Figura 2.9.

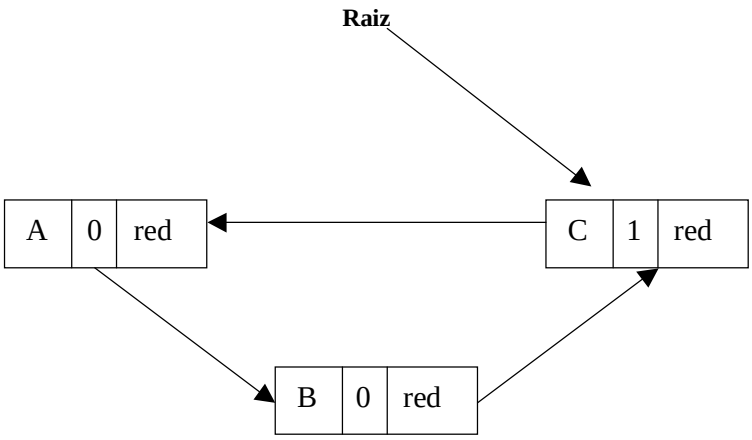


Figura 2.9 – Grafo logo após a chamada ao procedimento *Mark-red*

A Figura 2.9 mostra que a célula C ficou com seu contador maior que zero, indicando que ela tem uma referência externa, que no caso é a da raiz. O que ocorre, é que este fato só vai ser descoberto no final do *Scan*, que na prevendo a coleta, pintará as demais células de azul, como ilustra a Figura 2.10.

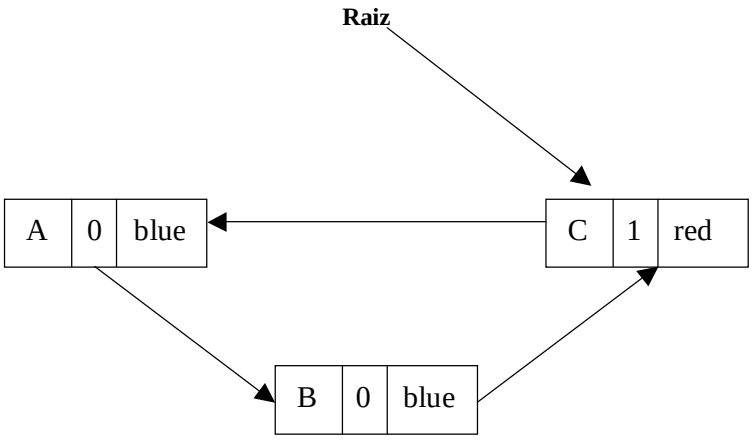


Figura 2.10 – Grafo logo após a chamada ao procedimento *Scan*

No final do *Scan*, ao encontrar a célula com uma referência externa, é feita a invocação ao procedimento *Scan-green*, que a partir desse ponto compartilhado, vai analisar as células que são vinculadas transitivamente à raiz e vai repintá-las de verde e reajustar seus contadores. A Figura 2.11 ilustra o cenário logo após o término do *Scan-green*.

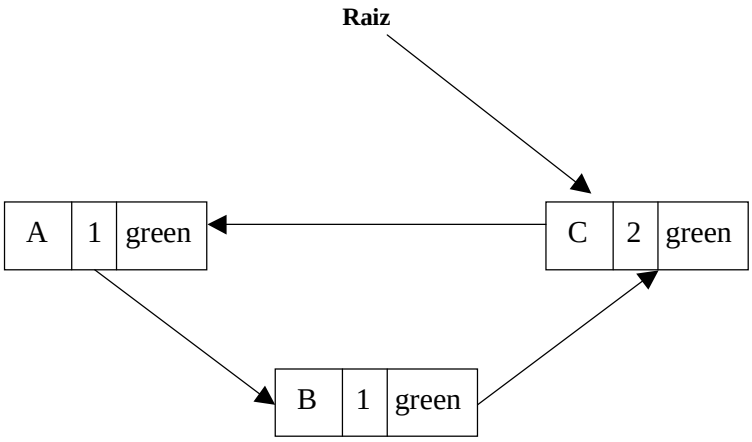


Figura 2.11 – Grafo logo após a chamada ao procedimento *Scan-green*

O problema é que logo em seguida o procedimento *Collect-blue* é invocado arbitrariamente, mas sem nenhum efeito. Isto torna o algoritmo estrito, pois ele sempre varre o sub-grafo toda vez que uma referência a uma célula compartilhada é excluída, tornando-o proibitivamente dispendioso e impraticável do ponto de vista de tempo consumido desnecessariamente **[JL96]**. Mesmo quando a coleta é bem sucedida o procedimento *Scan* sempre fará uma varredura completa na estrutura antes da chamada ao *Collect-blue*, o que vai totalizar a três varreduras do pior caso. Devemos observar ainda que quando o *Scan* encontra uma célula contendo uma referência externa, não significa que seu trabalho antecipado de pintar todas as células de azul tenha sido realizado em vão. Nesse cenário, pode ocorrer uma coleta parcial, pois parte do grafo pode não estar mais vinculado transitivamente à raiz. Observe a Figura 2.12. Se o ponteiro que vincula a raiz à célula A for excluído, as células A e D serão coletadas, enquanto que as células B, C, E e F permanecerão ativas.

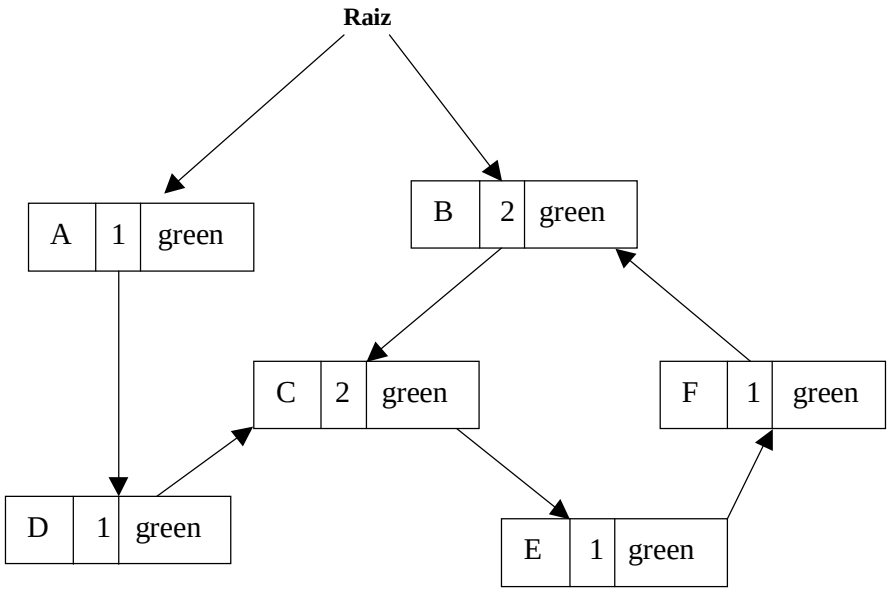


Figura 2.12 – Grafo onde poderá ocorrer uma coleta parcial

2.4.3.4 Algoritmo de contagem de referências cíclicas com *mark-scan lazy*

Lins **[Lin92]** estendeu o algoritmo original de contagem de referência cíclica utilizando a estratégia de postergar o *mark-scan* local, salvando as estruturas num conjunto de controle denominado ***Control-set.*** O *Control-set* é uma lista onde se guardam as referências às células compartilhadas removidas. Ela também é denominada *Control-queue*, sendo denotada algumas vezes pela letra *Q*.

Esta estratégia de procastinação do procedimento *mark-scan* é chamada *lazy* (estratégia preguiçosa), pois ela não realiza de imediato a varredura no sub-grafo no intuito de encontrar as células não mais referenciadas. Em algum momento propício, uma parte ou todo o *Control-set* é verificado para busca de lixo cíclico. O que diferencia este algoritmo do seu antecedente, é que, além das três cores, Lins utiliza cor preta (*black*), para indicar que a célula está presente no *Control-set*. Sempre que uma célula compartilhada é excluída, e ainda não estiver no *Control-set*, precisa ser pintada de preto e ter sua referência vinculada à estrutura. Esta quarta cor foi necessária para evitar referências duplicadas no conjunto. Posteriormente, Lins

[Lin02a] introduziu uma nova otimização deste algoritmo utilizando uma nova estrutura de dados que armazena os "pontos críticos" do grafo, aumentando a eficiência do algoritmo.

A Figura 2.13 ilustra o uso do *Control-set* após a exclusão do ponteiro <raiz,B>. A Célula B fica armazenada no *Control-set* e têm sua cor pintada de preto (*black*).

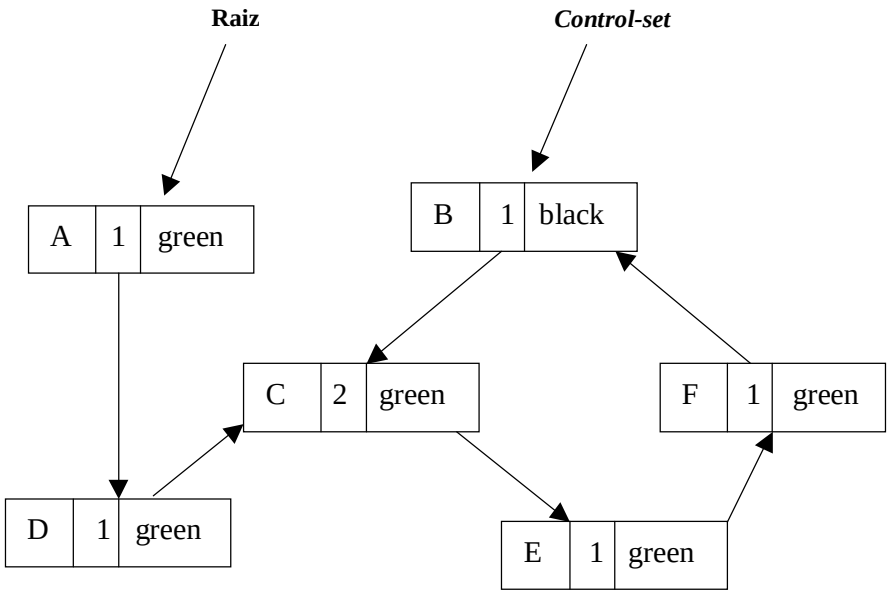


Figura 2.13 – Grafo após o comando *Delete*(<raiz,B>) com uso do *Control-set*

2.4.3.5 Algoritmo de contagem de referências cíclicas com *mark-scan lazy* - procedimentos

New(R): Retira uma célula U da *free-list* e cria o ponteiro <R,U>, onde R é a célula direta ou transitivamente vinculada à raiz. O *mark-scan* é realizado no *Control-set* através da chamada ao procedimento *Scan-control-set* quando as células da *free-list* se esgotam. O programa é abortado se não houver mais espaço na *heap*.

```
New(R) =
    if (free-list not empty) then
       select U from free-list
       make-pointer <R,U>
    else
       if (Control-set not empty) then
          Scan-control-set()
          New(R)
       else
          Abort "Memory exhausted"
```

Algoritmo 2.13 O procedimento *New* do algoritmo de contagem de referências cíclicas com *mark-scan lazy*

Copy(R, <S,T>): Cria um ponteiro <R,T>, onde R e S são células transitivamente vinculadas à raiz e <S,T> é um ponteiro do grafo. Incrementa o contador da célula alvo e ajusta sua cor para verde (*green*).

```
Copy(R, <S,T>) =
   Make-pointer <R,T>
   Increment RC(T)
   colour(T) = green
```

Algoritmo 2.14 O procedimento *Copy* do algoritmo de contagem de referências cíclicas com *mark-scan lazy*

Delete(<R,S>): É idêntico ao algoritmo estrito, diferenciando no fato que o procedimento *mark-scan* é realizado de forma *lazy*. A cor da célula é sempre testada para que não haja referências redundantes na estrutura para que depois decida sob sua inclusão. Quando o *Control-set* tem seu limite alcançado, é invocado o procedimento *Scan-control-set* para liberar espaço.

```
Delete(<R,S>) =
   Remove <R,S>
   if (RC(S) == 1) then
      for T in Sons(S) do
         Delete(S,T)
      Link-to-free-list(S)
   else
      Decrement RC(S)
      if (colour(S) != black) then
          if (Control-set is full) then
             Scan-control-set()
          else
             colour(S) = black
             Link-to-Control-set(S)
```

Algoritmo 2.15 O procedimento *Delete* do algoritmo de contagem de referências cíclicas com *mark-scan lazy*

Scan-control-set(): Realiza o *mark-scan* local. O algoritmo toma uma célula e verifica sua cor. Se ela permanece preta, precisa ser analisada pelo *mark-scan* local; caso contrário, ela é removida do *Control-set*. Deve-se lembrar também que o *Delete* pode a ter enviado direto para a *free-list* ou o *Copy* a identificou como ativa.

```
Scan-control-set() =
   S = head(Control-set)
   Control-set = tail(Control-set)
   if (colour(S) == black) then
       Mark-red(S)
       Scan(S)
       Collect-blue(S)
   else
      if (Control-set not empty) then
         Scan-control-set()
```

Algoritmo 2.16 O procedimento *Scan-control-set* do algoritmo de contagem de referências cíclicas com *mark-scan lazy*

Mark-red(S): É idêntico ao do algoritmo estrito, diferindo apenas no fato de que a célula pode ser preta além de verde, ou seja, é diferente de vermelho.

```
Mark-red(S) =
   if (colour(S) != red) then
      colour(S) = red
      for T in Sons(S) do
         Decrement RC(T)
         if (colour(T) != red) then
            Mark-red(T)
```

Algoritmo 2.17 O procedimento *Mark-red* do algoritmo de contagem de referências cíclicas com *mark-scan lazy*

Os cenários para o algoritmo *lazy* que utiliza o *Control-set* são os mesmos, apenas considera-se que o processamento do *mark-scan* local é feito sob demanda e não logo após a exclusão de uma célula compartilhada. Para não se tornar repetitivo, analisaremos os exemplos das seções 2.4.3.7 e 2.4.3.8 os cenários do algoritmo estrito e *lazy* com o uso do *Control-set* em conjunto com uma outra estrutura de otimização, a *Jump-stack*.

2.4.3.6 Algoritmo de contagem de referência cíclica com o uso da *Jump-stack*

Lins **[Lin02a]** introduziu uma nova estrutura de dados, uma pilha, denominada *Jump-stack*, que armazena referências a "pontos críticos" do grafo durante a fase de marcação local após a exclusão de uma referência para uma célula compartilhada. O novo algoritmo em vez de retornar ao início do *mark-scan*, ele "salta" diretamente o grafo, analisando os pontos críticos de imediato, evitando chamadas desnecessárias ao procedimento *Scan*.

O algoritmo funciona em duas fases. Na primeira fase, o sub-grafo da referência removida é percorrido para ajustar os contadores para que espelhem apenas as referências externas e as células são pintadas de vermelho, ficando marcadas como já visitadas. Toda vez que uma célula, que tem seu contador de referência diferente de zero, for excluída, fará com que o algoritmo armazene na *Jump-stack* todas as células que potencialmente estão transitivamente vinculadas à raiz. Na segunda fase, as células apontadas pelas referências contidas na *Jump-stack* são visitadas de forma direta. Se uma célula que está sendo analisada permanecer com o contador maior que zero, o sub-grafo inteiro sob ela está em uso e precisa ter os contadores de suas células reajustados. Na parte final dessa fase, será realizada a coleta das células não mais referenciadas. Não é mais necessário percorrer as células lixo pintadas de

azul. O lixo que eventualmente existir será recolhido de forma direta pelo procedimento *Collect* depois do processamento da *Jump-stack*. Neste caso, a fase de *Scan* é poupada o máximo possível.

2.4.3.7 Algoritmo de contagem de referência cíclica com o uso da *Jump-stack* - procedimentos

Delete(<R,S>): É idêntico ao algoritmo estrito, diferenciando no fato que os procedimentos *Mark-red* e *Scan* são invocados nas últimas linhas do algoritmo.

```
Delete(<R,S>) =
   Remove <R,S>
   if (RC(S) == 1) then
      for T in Sons(S) do
         Delete(<S,T>)
      Link-to-free-list(S)
   else
      Decrement RC(S)
      Mark-red(S)
      Scan(S)
```

Algoritmo 2.18 O procedimento *Delete* do algoritmo de contagem de referências cíclicas com o uso da *Jump-stack*

Mark-red(S): Em seu código que está embutido a otimização do algoritmo com o uso da *Jump-stack*. Armazenando as possíveis referências externas ao sub-grafo, evitando-se assim que o procedimento *Scan* percorra uma área de lixo para pintar as células de azul desnecessariamente. Com o processamento da *Jump-stack*, se restar alguma célula, é lixo que pode ser coletado diretamente.

```
Mark-red(S) =
   if (colour(S) == green) then
      colour(S) = red
      for T in Sons(S) do
         Decrement RC(T)
         if (RC(T) > 0 && T not in Jump-stack) then
            Push-Jump-stack(T)
         if (colour(T) == green) then
            Mark-red(T)
```

Algoritmo 2.19 O procedimento *Mark-red* do algoritmo de contagem de referências cíclicas com o uso da *Jump-stack*

Scan(S): Verifica se a *Jump-stack* tem algum elemento. Se estiver vazia, as células sob S são vinculadas à *free-list* por meio do procedimento *Collect*. Se existir algum elemento na *Jump-stack*, significa que ainda existem células no sub-grafo a serem analisadas. Se estas células têm seus contadores de referências maiores que zero, então há ainda referências externas que as vinculam transitivamente à raiz e estes contadores têm que ser reajustados. Este reajuste é realizado pelo procedimento *Scan-green*, que continua sendo o mesmo.

```
Scan(S) =
   if (RC(S) > 0) then
      Scan-green(S)
   else
       while (Jump-stack not empty) do
          T = top-of-Jump-stack
          Pop-Jump-stack
          if (colour(T) == red && RC(T) > 0) then
             Scan-green(T)
   Collect(S)
```

Algoritmo 2.20 O procedimento *Scan* do algoritmo de contagem de referências cíclicas com o uso da *Jump-stack*

Collect(S): Verifica células de cor vermelho (células lixo) e as retorna para a *free-list*.

```
Collect(S) =
   if (colour(S) == red) then
      for T in Sons(S) do
          Remove(<S,T>)
          RC(S) = 1
          colour(S) = green
          Link-to-free-list(S)
          if (colour(T) == red) then
             Collect(T)
```

Algoritmo 2.21 O procedimento *Collect* do algoritmo de contagem de referências cíclicas com o uso da *Jump-stack*

Um melhoramento neste algoritmo ainda foi obtido otimizando-se o uso da *Jump-stack*, de forma a postergar a inserção de referências e, com isso, garantir uma única tentativa de inserção para cada elemento do sub-grafo. Esta otimização está incluída no procedimento *Mark-red*, de modo que na chamada de S, sua inserção só é definida após a chamada recursiva do procedimento para seus filhos em *Sons*(S). Desta forma, a decisão de incluir um elemento na *Jump-stack* é procastinada e, se esta inclusão acontecer, será realizada uma única vez. Observe o Algoritmo 2.22 do procedimento *Mark-red*.

```
Mark-red(S) =
   if (colour(S) == green) then
      colour(S) = red
      for T in Sons(S) do
          Decrement RC(T)
          if (colour(T) == green) then
             Mark-red(T)
      if (RC(S) > 0) then
         Push-Jump-stack(T)
```

Algoritmo 2.22 Novo procedimento *Mark-red* do algoritmo de contagem de referências cíclicas com o uso otimizado da *Jump-stack*

Nos cenários dos algoritmos com o *mark-scan* estrito e *lazy* com a *Jump-stack*, analisaremos novamente duas situações onde o *mark-san* local é processado. A diferença entre o algoritmo estrito e o *lazy*, reside no fato de que o segundo posterga o processamento local através do uso do *Control-set*.

No algoritmo *lazy*, quando se executa o procedimento S*can-control-set*, o *mark-can* utiliza a *Jump-stack* do mesmo modo que o estrito. Com isso, os cenários analisados a seguir, são válidos para os dois casos com a *Jump-stack*.

No primeiro caso, mostraremos o comportamento do algoritmo na estrutura mínima do grafo mostrada na tese de mestrado de Salkild **[Sal87]** que encontrou uma falha no algoritmo de contagem de referências cíclicas proposto por Brownbridge **[Bro85]** em 1985. Nessa estrutura, não há células a serem reclamadas quando um ponteiro que vincula a raiz a uma célula é excluído. No segundo caso, mostraremos o comportamento do algoritmo quando os dois ponteiros que ligam a raiz à estrutura são excluídos, fazendo com que as células do grafo que formam o ciclo sejam vinculadas à *free-list* através do procedimento *mark-scan* local que utiliza a *Jump-stack* para evitar as chamadas recursivas desnecessárias ao *Scan*.

2.4.3.8 Cenário do primeiro caso: nenhuma estrutura é recuperada

Neste cenário, mostraremos o comportamento do algoritmo com a estrutura de Salkild, que é ilustrada na Figura 2.14, onde nenhuma célula é reclamada após a exclusão de um ponteiro de uma célula compartilhada que possui referência externa. O ponteiro que liga a célula A à célula B será excluído através da chamada ao procedimento *Delete*(A,B).

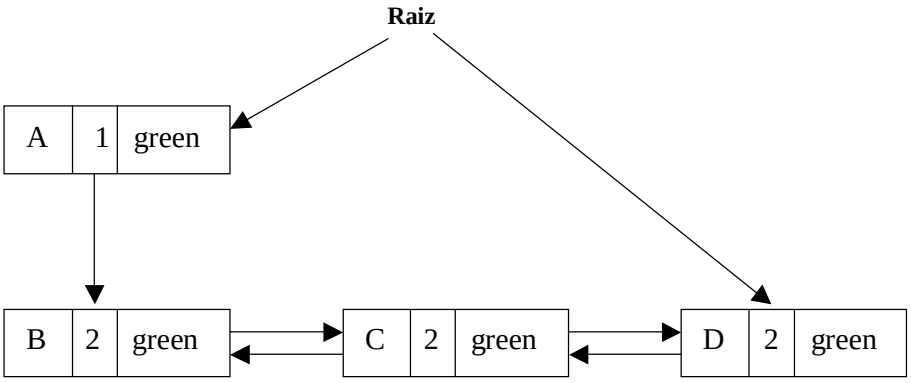


Figura 2.14 – Grafo do cenário inicial de Salkild

Na Figura 2.15, com a exclusão do ponteiro <A,B> pelo *Delete*, a célula B é identificada como uma célula compartilhada, uma vez que tem o contador maior que um. Com isso, um *mark-scan* local vai ser acionado para realizar a varredura - o mesmo caso se aplica ao algoritmo *lazy* quando o *Control-set* se encontra cheio e a célula B é a primeira a ser examinada. Então, o contador de B sofre um decréscimo de uma unidade através da execução da primitiva "Decrement RC(B)" do Delete, e logo em seguida os procedimentos *Mark-red e Scan* são acionados. No *Mark-red*, B é pintado de vermelho (*red*) e o contador do seu filho C sofre um decréscimo e fica com o valor um. Como a célula C tem seu contador maior que zero, ela é vinculada à *Jump-stack* (desde que não tenha sido incluída anteriormente para que se evite a repetição de uma mesma célula dentro dessa estrutura).

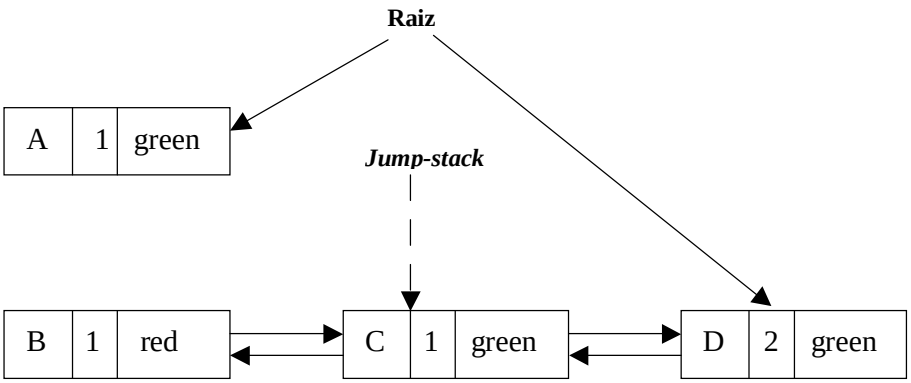


Figura 2.15 – Grafo depois da execução do *Mark-red* em B

A ação recursiva do *Mark-red* continua atuando, desta vez sobre C, que passa a ter a cor vermelha e o contador de B cai a zero. O contador de D também é atualizado e cai para um e, com isso, é vinculado à *Jump-stack*, como fica ilustrado na Figura 2.16.

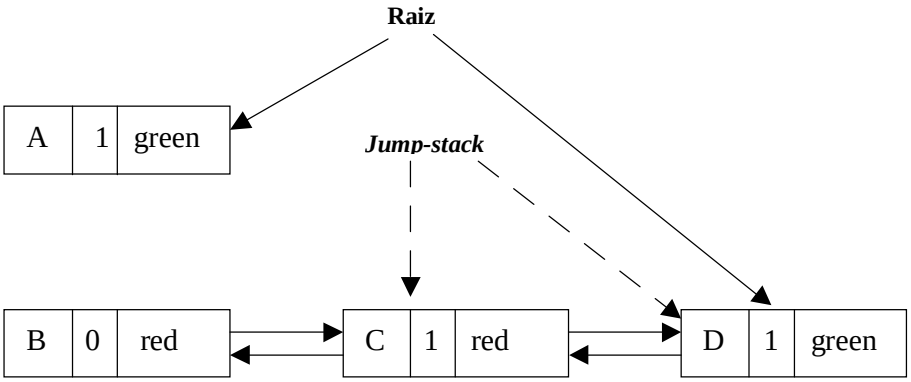


Figura 2.16 – Grafo depois da execução do *Mark-red* em C

Seguindo sua execução, o *Mark-red* atua sobre D, que passa a ter a cor vermelha e contador de C é atualizado para zero, como ilustra a Figura 2.17. A figura também mostra que todas as células do sub-grafo cíclico estão pintadas de vermelho. Com isso o procedimento *Mark-red* cessa sua execução. Observe que a *Jump-stack* guardou as possíveis referências a pontos críticos do grafo que o conectam direta ou transitivamente à raiz.

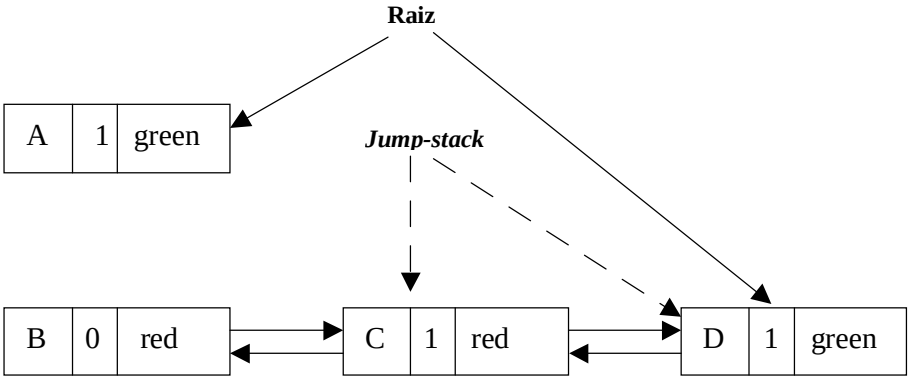


Figura 2.17 – Grafo depois da execução do *Mark-red* em D

É a vez do procedimento *Scan* iniciar seu processamento em B. É neste procedimento que verificamos a vantagem do uso da *Jump*-stack, onde há duas possibilidades de processamento. Na primeira, se o contador de B for maior que zero, a *Jump-stack* é esvaziada e o procedimento *Scan-green* é acionado logo em seguida. Na segunda, a *Jump-stack* tem que

ser verificada. Como no nosso caso o contador de B é zero, então a *Jump-stack* vai ser utilizada no processamento. Seu topo é capturado, que no caso é D, e seu contador vai ser analisado. Se for zero, D será rejeitado e o próximo elemento da estrutura é capturado, que no caso é C. Se for maior que zero, o procedimento *Scan-green* é acionado. A Figura 2.18 ilustra o grafo após o processamento do *Scan-green*, onde todas as células tiveram seus contadores reajustados, além de terem sido repintadas de verde.

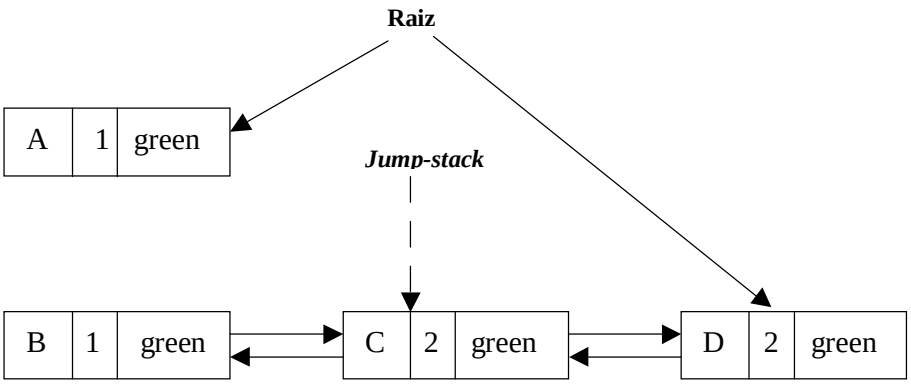


Figura 2.18 – Grafo depois da execução do *Scan-green*

O processamento do *Scan* prossegue analisando a célula C capturada da *Jump-stack*, mas como ela já foi repintada de verde pelo *Scan-green*, ela é ignorada. Mesmo assim, o procedimento *Collect* é acionado em B, mas também não causa feito algum, pois não há nenhuma célula a coletar. A Figura 2.19 ilustra a configuração final do grafo.

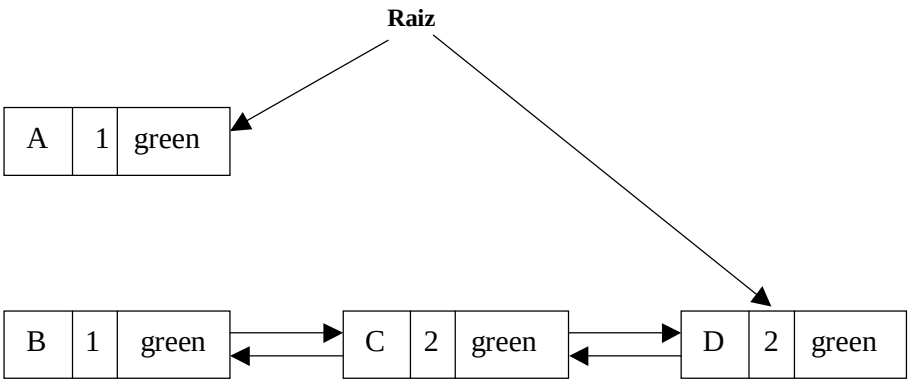


Figura 2.19 – Grafo depois do processamento da *Jump-stack* no *Scan*

2.4.3.9 Cenário do segundo caso: estrutura cíclica reclamada

Para ilustrar esse cenário, utilizaremos a mesma configuração da Figura 2.12 com apenas uma pequena mudança: o ponteiro da célula C para D, e não o contrário. A Figura 2.20 ilustra a situação inicial do grafo, onde a remoção do ponteiro que vincula célula B à raiz causará o isolamento de um ciclo em relação à própria raiz.

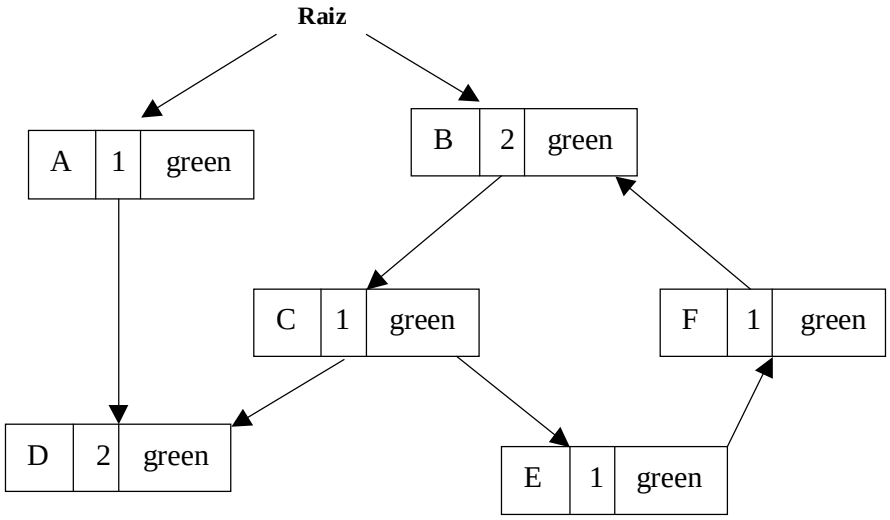


Figura 2.20 – Grafo onde ocorrerá a reclamação de um ciclo

A primeira ação do procedimento *Delete*, ao se remover o ponteiro <Raiz,B>, é atualizar o contador de B para um através da chamada à primitiva "Decrement RC(B)". Logo em seguida, os procedimentos M*ark-red e Scan* são acionados sobre B. O *Mark-red* atua em B, pintando-a de vermelho e reduzindo o contador de C para zero. A Figura 2.21 ilustra a nova configuração do grafo.

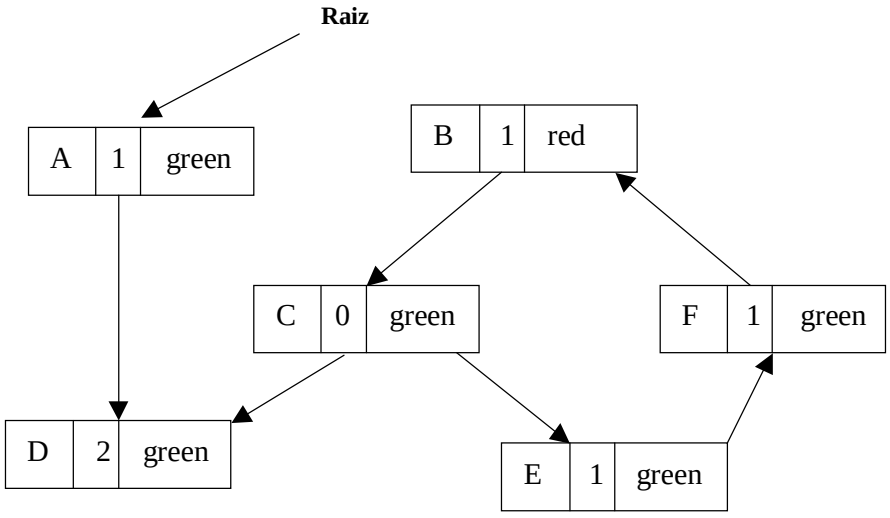


Figura 2.21 – Grafo após *Mark-red* em B

O *Mark-red* atua em C, pintando-a de vermelho e reduzindo os contadores de seus filhos D e E, para um e zero, respectivamente. Como a célula D ficou com o contador maior que zero, então ela é adicionada à *Jump*-stack. A Figura 2.22 ilustra a nova configuração do grafo.

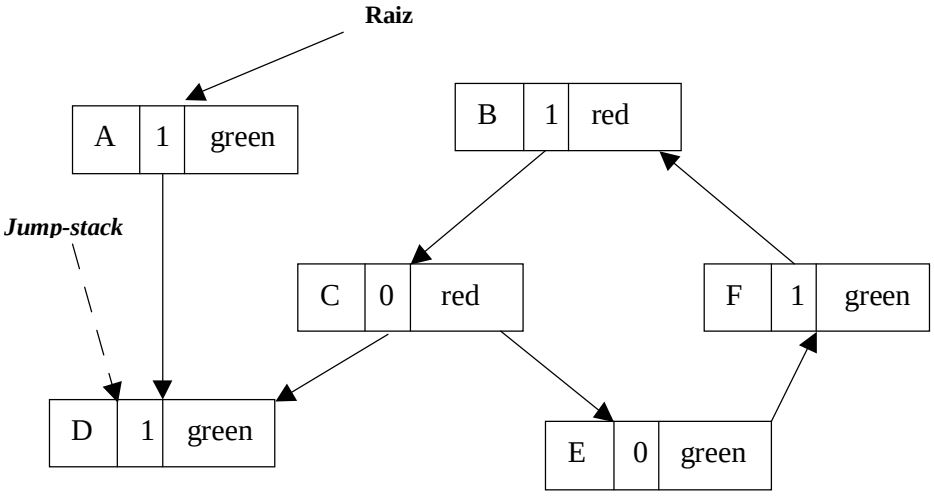


Figura 2.22 – Grafo após *Mark-red* em C

O *Mark-red* atua em D, pintando-a de vermelho e, como ele não tem filhos, o procedimento salta recursivamente para E, pintando-a de vermelho e reduzindo o contador de F para zero. A Figura 2.23 ilustra a nova configuração do grafo.

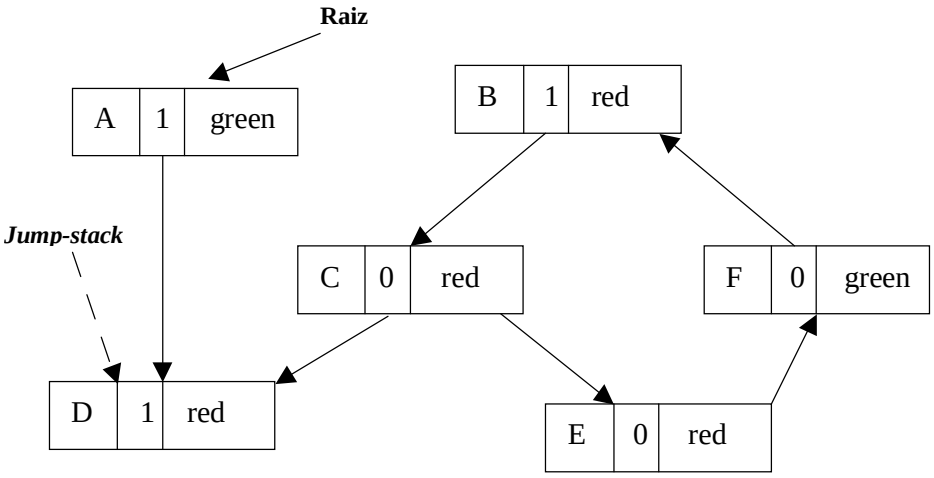


Figura 2.23 – Grafo após *Mark-red* em D e em E

Seguindo recursivamente, o *Mark-red* atua em F, pintando-a de vermelho e contador de B cai para zero. Com isso encerra-se o ciclo e o sub-grafo agora tem todas as suas células pintadas de vermelho, como ilustra a Figura 2.24.

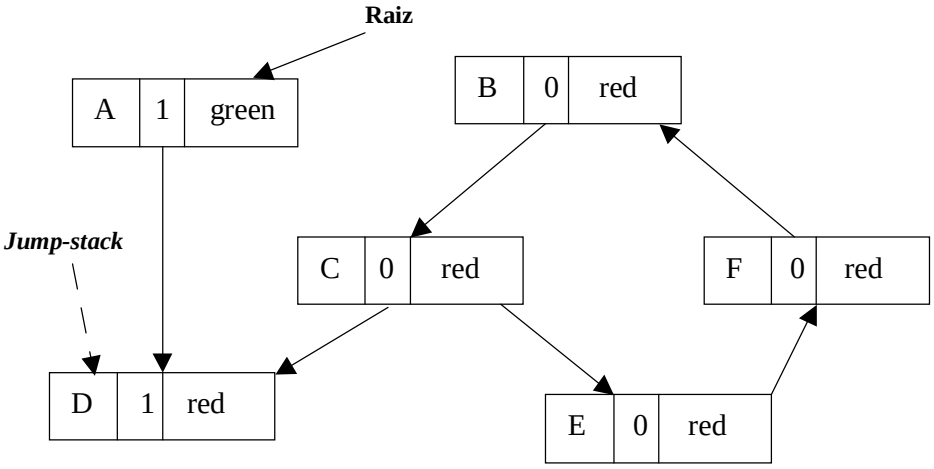


Figura 2.24 – Grafo após o término do *Mark-red* em F

Terminado o procedimento *Mark-red*, o procedimento *Scan* entra em ação logo em seguida a partir do ponto de início da recursividade, ou seja, a partir da célula B. o *Scan* verifica se tem algum contador maior que zero. Como não tem, parte para o processamento da *Jump-stack.* A célula D é capturada e tem sua cor restaurada para verde, como ilustra a Figura 2.25.

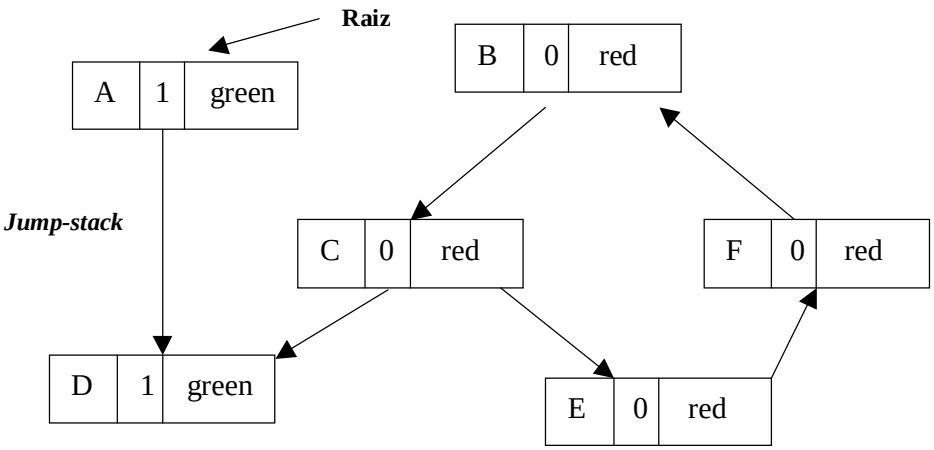


Figura 2.25 – Grafo após o processamento da *Jump-stack*

Logo após o processamento da *Jump-stack*, entra em ação o processamento do *Collect-blue*. Ele atua recursivamente, a partir de B, sobre o sub-grafo marcado de vermelho, reclamando todas as suas células para a *free-list*. Todas as células do sub-grafo têm suas cores atualizadas para verde e seus contadores atualizados para um, indicando que as células estão disponíveis para re-uso. Note também que a célula C foi desvinculada da célula D, antes de ser incluída na *free-list,* e que também o ciclo do sub-grafo é desfeito.

A configuração final do grafo é ilustrada na Figuar 2.26.

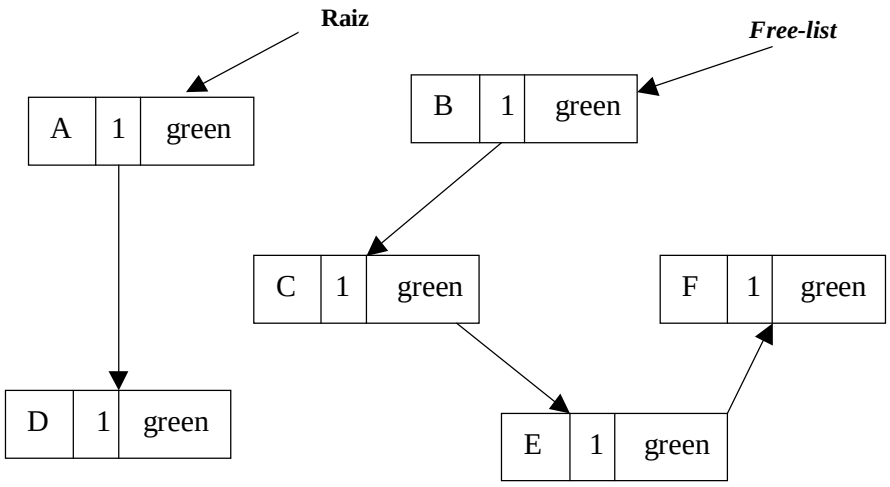


Figura 2.26 – Grafo após o processamento do *Scan*, na sua configuração final

Podemos agora analisar o cenário do algoritmo com o uso otimizado da *Jump-stack* no *Mark-red* (Algoritmo 2.24). A Figura 2.27 ilustra a configuração inicial do cenário em que essa utilização otimizada torna-se clara.

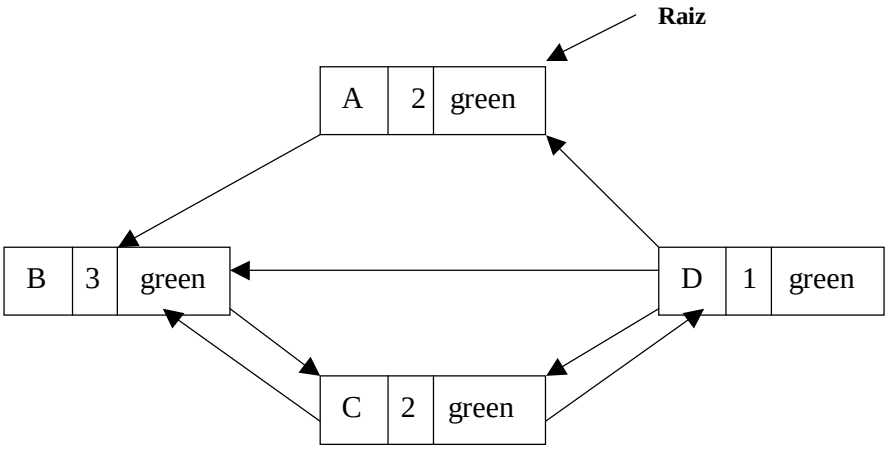


Figura 2.27 – Grafo para uso otimizado da *Jump-stack*

A exclusão do ponteiro que vincula a raiz à célula A, faz com que seu contador seja atualizado para um. Logo em seguida entra em ação o procedimento *Mark-red* em A. que a pinta de vermelho e atualiza o contador de B para dois, ficando o grafo conforme ilustrado na Figura 2.28.

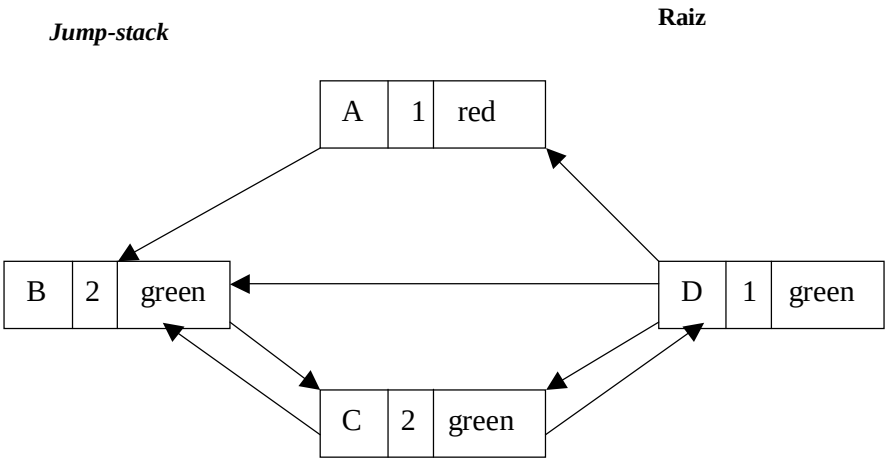


Figura 2.28 – Grafo após a exclusão do ponteiro <Raiz,A>

Note que no algoritmo com uso não otimizado da *Jump-stack*, a célula B seria inserida nela, pois seu contador é maior que zero. No entanto, sua inclusão é postergada devido a chamada recursiva do procedimento.

O procedimento *Mark-red* continua sua execução recursiva agora em B, que tem sua cor atualizada para vermelho e o contador de C é atualizado para um, como ilustra a Figura 2.29. Como no caso anterior, C continua com seu contador maior que zero, mas sua inclusão na *Jump-stack* é adiada.

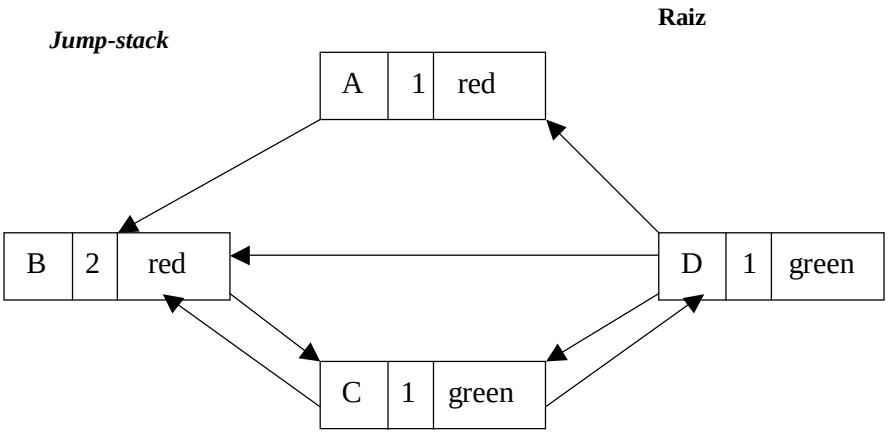


Figura 2.29 – Grafo após o *Mark-red* em B

O *Mark-red* prossegue recursivamente para C, onde sua cor é atualizada para vermelho, o contador de B cai para um e o de D para zero, como ilustra a Figura 2.30. Observe que se não fosse o uso otimizado da *Jump-stack*, B novamente seria testada para sua inserção nessa estrutura.

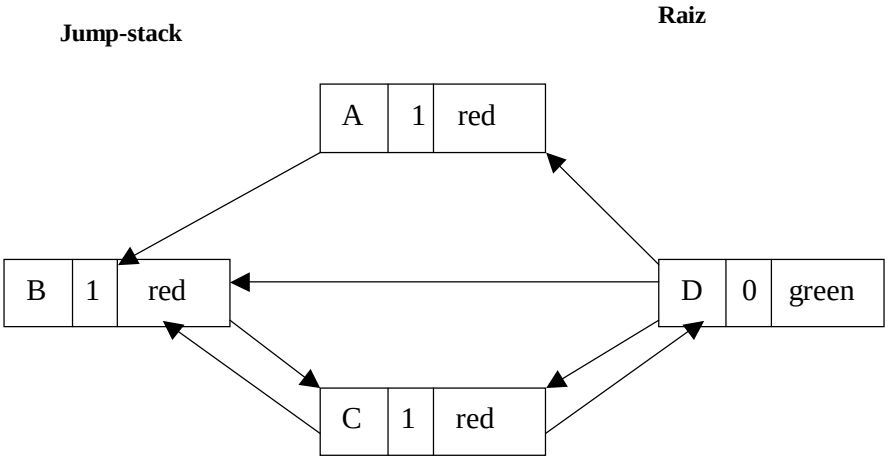


Figura 2.30 – Grafo após o *Mark-red* em C

Por último, o *Mark-red* é executado em D, onde sua cor é atualizada para vermelho e seus filhos A e B, além de C, têm seus contadores decrementados para zero, como ilustra a Figura 2.31. Como B está com seu contador zerado, ela não é inserida na *Jump-stack*.

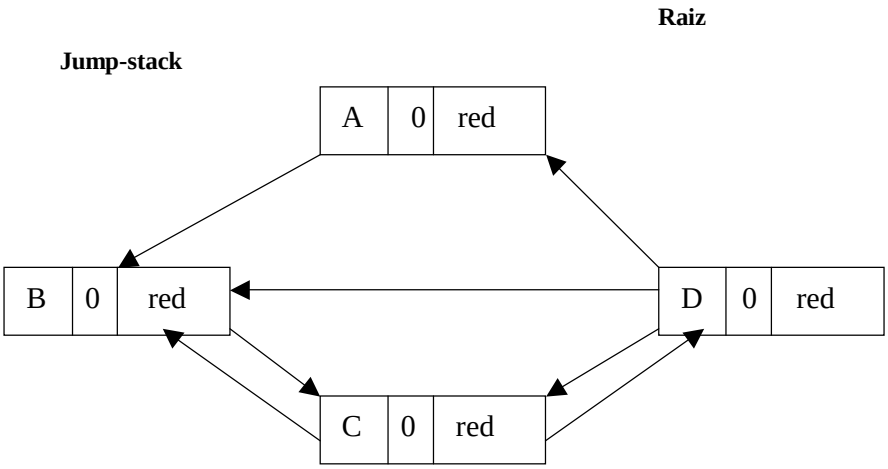


Figura 2.31 – Grafo após o *Mark-red* em D

Uma descrição comparativa entre os algoritmos de contagem de referências cíclicas pode ser encontrada em **[Sal02]**.

2.4.4 Coleta por cópia

O algoritmo de *mark-scan* consome tempo visitando células em uso (na marcação) e depois varre toda a memória. Tal procedimento é otimizado pelo algoritmo de cópia. O primeiro algoritmo de coleta por cópia foi desenvolvido por Marvin Minsky **[Mar63]** para uso na linguagem LISP versão 1.5, que copiava as células vivas da *heap* para uma área de armazenamento secundário - fita magnética - e depois as revertia de volta para a mesma *heap*.

Com o advento dos sistemas operacionais com memória virtual, Fenichel e Yochelson **[FY69]** "ressuscitaram" o algoritmo de Minsky. O algoritmo de cópia de Fenichel e Yochelson divide o espaço da *heap* em dois semi-espaços contíguos denominados de Espaço de Origem e Espaço de Destino (*FromSpace* e *ToSpace*). O algoritmo inicia o revezando (*flipping*) dos semi-espaços, onde o Espaço de Origem torna-se o Espaço de Destino e vice-versa. Então, cada objeto no Espaço de Origem é copiado no Espaço de Destino, juntamente com todos seus descendentes. A fim de evitar cópias múltiplas do mesmo objeto que podem ser alcançados por caminhos múltiplos, um ponteiro guia é instalado dentro da versão anterior do objeto. Quando o processo de busca encontra o ponteiro dentro do Espaço de Origem, o

objeto referido é checado para ver o ponteiro guia. Se houver um, ele estará pronto para ser movido para o Espaço de Destino, assim como seu ponteiro alcançado terá simplesmente atualizado sua nova posição.

Todos os objetos vivos serão eventualmente copiados para o Espaço de Destino e o lado bom desta abordagem é que ela resolve o problema de fragmentação da memória de modo natural.

O problema principal da coleta de cópia é que ela reduz pela metade o tamanho disponível de armazenamento efetivo. Além disso, é uma abordagem pare-e-colete.

A Figura 2.32 ilustra o esquema deste algoritmo.

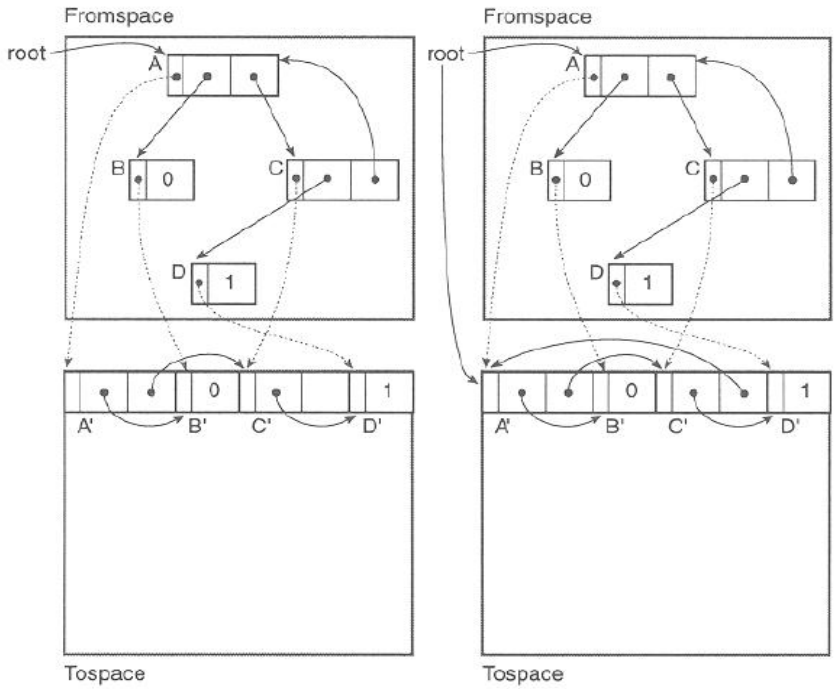


Figura 2.32 – Esquema da *heap* na coleta por cópia **[JL96]**

2.4.4.1 Algoritmo coleta por cópia - procedimentos

Neste algoritmo, há quatro atualizações no grafo: A inicialização, através do procedimento *Init*; criação, através do procedimento *New*; a reversão dos espaços, através do procediemnto *Flip*; e a cópia do espaço, através do procedimento *Copy*.

Init(): É o ponto inicial do algoritmo que faz a divisão da *heap* em dois semi-espaços.

```
Init() =
   Tospace = heap-bottom
   Space-size = heap-size / 2
   Top-of-space = Tospace + Space-size
   free = Tospace
```

Algoritmo 2.23 O procedimento *Init* no algoritmo de cópia

New(n): Verifica se há espaço suficiente para apontar para o próximo bloco livre e aloca a nova célula.

```
New(n) =
   Top-of-space to Tospace + Space-size
   if free + n > Top-of-space
       Flip()
   if free + n > Top-of-space
       Abort “memory exhausted”
   newcell = free
   free = free + n
   return newcell
```

Algoritmo 2.24 Alocação no algoritmo de cópia

Flip(): Caso não haja espaço suficiente para alocação, o mutador pára e a coleta de lixo é iniciada pelo *Flip*, que inverte os papéis de *Espaço de Origem* e *Espaço de Destino*.

```
Flip() =
   Tempspace = Fromspace
   Fromspace = Tospace
   Tospace = Tempspace
   Top-of-space = Tospace + Space-size
   free = Tospace
   for R in Roots
       R = Copy(R)
```

Algoritmo 2.25 O *Flip* no algoritmo de cópia

Copy(P): A versão deste pseudocódigo é do algoritmo de cópia desenvolvido por Fenichel e Yochelson **[FY69]**.

```
/* Obs: P aponta para uma palavra, não uma célula */
Copy(P) =
   if atomic(P) or P == nil   /* P não é um ponteiro */
      Return P
   if not forwarded(P)
      n = size(P)
      P' = free    /* reserva espaço em Tospace */
      free = free + n
      temp = P[0]   /* campo 0 é o início do endereço */
      forwarding_address(P) = P'
      P'[0] = copy(temp)
      for i = 1 to n-1   /* copia cada campo de P para P' */
         P'[i] = copy(P[i])
   Return forwarding-address(P)
```

Algoritmo 2.26 O algoritmo de cópia de Fenichel-Yochelson

Um problema que deve ser levado em consideração neste algoritmo, é que células compartilhadas podem ser copiadas mais de uma vez devido ao compartilhamento e, com isso, provocar uma situação crítica se o espaço de origem contiver ponteiros que formam ciclos. Também múltiplas cópias de um mesmo objeto poderiam aumentar a ocupação da *heap*, levando a uma quebra semântica do aplicativo do usuário. Minsky **[Min63]** solucionou este problema utilizando indireções (*forwarding address*) no *Espaço de Origem*. O algoritmo utiliza um bit de marca extra em cada célula para ser usado na durante a varredura e

marcação. Os conteúdos de seus campos são copiados para o novo endereço da célula. Esta indireção é depois colocada na célula marcada para que toda vez que um ponteiro fizer referência a ela, esse ponteiro será ajustado para refletir a transposição.

Segundo **[JL96]**, o algoritmo de cópia mais utilizado é o de C.J. Cheney **[Che70]**, que é iterativo ao eliminar do algoritmo de Fenichel-Yochelson, o custo das chamadas recursivas, com o uso uma fila, e o risco de *overflow* na pilha de execução. Ele utiliza o próprio *Espaço de Destino* para o armazenamento nas células que foram copiadas, ao invés de usar memória adicional para a fila.

2.4.5 Coleta de lixo generacional

O algoritmo de cópia despende um grande espaço copiando células em uso entre semi-espaços a cada coleta. A coleta generacional evita cópia de objetos longevos. A hipótese generacional assume que quanto mais antiga a célula maior as suas chances de "sobreviverem" a outras coletas.

Neste algoritmo, a *heap* é dividida em partes ou gerações. Experimentos mostraram que a divisão em apenas duas gerações atinge os melhores resultados. A geração velha contém todos os objetos que tem o tempo de permanência alto e a geração nova contém os objetos recém-criados. A geração nova sofre coleta de lixo por cópia mais freqüentemente que a geração velha.

Para nosso estudo, esta técnica não tem uma técnica correspondente no campo da coleta de lixo distribuído, e por isso não a descreveremos em detalhes aqui.

Capítulo 3

Algoritmos de coleta de lixo em ambiente distribuído

Como a World Wide Web é por natureza um ambiente distribuído, é de nosso interesse agora estudar os algoritmos de coleta de lixo nesse ambiente. Os algoritmos de coleta de lixo distribuído são “adaptações” dos algoritmos clássicos das arquiteturas centralizadas apresentadas no capítulo anterior.

O algoritmo para o gerenciamento consistente de páginas Web, ápice desta dissertação, que será apresentado no próximo capítulo é uma aplicação do algoritmo de Lins **[Lin02]** para ambiente distribuído. Estabeleceremos um paralelo entre páginas Web e “células” de informação em um sistema distribuído com o coletor de lixo. Dessa maneira, faz-se necessário apresentar o funcionamento básico dos algoritmos para o gerenciamento de memória distribuída, de forma a possibilitar o entendimento da conveniência e adequação do algoritmo de Lins **[Lin02]** como ponto de partida.

3.1 Redes de computadores e a coleta de lixo

Sistemas distribuídos podem ser vistos em dois níveis. O nível mais baixo é o de redes de computadores e o mais alto o nível de protocolos de distribuição que criam a memória virtual compartilhada **[JL96]**. Segundo **[Tan03]**, computadores que estão delimitados por uma pequena distância formam uma rede de computadores locais (**LAN**). Por outro lado, uma rede espalhada geograficamente forma uma rede de longa distância (**WAN**).

A comunicação nas LANs são bem mais rápidas e eficientes devido a curta distância e à tecnologia utilizada. As máquinas podem estar conectadas por fibras óticas, permitindo uma transmissão na faixa de centenas de *megabits* de dados por segundo. Pode haver uma

computação distribuída real envolvendo paralelismo bem afinado entre os processadores a fim de que juntos possam chegar a uma solução de um problema modelado. Em uma LAN podemos assumir que há possibilidade de alguma classificação global de sincronismo possa e ocorrer e que as estruturas de dados envolvidas tendem a ser menores e interdependentes. A migração passiva de objetos entre processadores para aumentar a localidade dos dados pode ser tecnicamente viável em uma LAN. A coleta de lixo em uma LAN pode ser vista com uma extensão complexa do ambiente de um processador, ou mais apropriadamente, como técnicas de memória compartilhada. Embora não seja explicitamente declarado, pode-se afirmar que todos algoritmos existentes atualmente para coleta de lixo distribuída são voltados para LANs. **[JL96]**.

Nas WANs, a comunicação não é confiável, além de ser dispendiosa. Mensagens podem perdidas, corrompidas, duplicadas, atrasadas por seguirem caminhos diferentes ou podem ser recebidas fora da ordem da qual foram enviadas. Os custos de comunicação e a segurança de dados podem impor a restrição de se ter informações replicadas a fim de aumentar a localidade e com isso reduzir a latência de acesso aos dados e aumentar a confiabilidade. Enquanto as paradas num processamento distribuído para que o coletor de lixo atue é aceitável em uma LAN, numa WAN isto é intoleravelmente ineficiente. Em uma WAN, os processos e processadores são autônomos e se conectam para um fim específico, possivelmente para realizar uma atividade de curta duração, tal como uma consulta remota em um banco de dados distribuído. Um objeto ou processo que não é transitivamente conectado ao grafo de processos ativos ou células, pode vir a se tornar ativo através de um envio de mensagem para outro objetivo ativo. Um problema é que mensagens podem ficar pendentes em uma rede e o assincronismo em uma WAN torna muito difícil determinar um processo (ou célula) topologicamente em um dado momento. Um algoritmo de coleta de lixo distribuído deve levar em consideração todos esses fatores **[JL96]**.

A coleta de lixo em sistemas distribuídos começou a ser aplicada visando transparência na reclamação de estruturas dinâmicas. Surgiram várias propostas de implementação como a de Hughes **[Hug85]** e Lang et al. **[LQP92]**. Bevan **[Bev87]** e Watson & Watson **[WW87]** que apresentaram uma proposta que envolvia um esquema de arquitetura de sistemas paralelos baseada na contagem de referência ponderada. Talvez a primeira descrição de um esquema de contagem de referência ponderada seja creditada a Weng **[Wen79]**.

3.2 Por que a coleta de lixo distribuída é diferente?

A principal diferença entre os coletores distribuídos e centralizados (arquiteturas de monoprocessadores ou de memória compartilhada) está na presença de mecanismos de passagem de mensagem utilizados na comunicação entre processadores.

Comunicação baseada em mensagem (sobre conexões potencialmente não confiáveis) traz a tona as seguintes questões específicas dos coletores distribuídos:

- Como evitar condições de competição? A ordem dos eventos é algumas vezes importante. Isto pode ser importante quando duas requisições enviadas pelo nó para dois diferentes nós sejam processadas na mesma ordem em que foram emitidas. Note-se que isto é um problema que pode ocorrer mesmo se a comunicação é livre de erro. Um cenário simples mostrando como condições de competição podem ocorrer (afetando a propriedade de segurança e, conseqüentemente a exatidão do coletor) é apresentando posteriormente nesta seção.
- Um novo tipo de custo. O mecanismo de passagem de mensagem introduz o custo de comunicação entre os processadores.
- Escalabilidade. Qualquer coletor distribuído deve escalar o tanto quanto o número de nós crescer no sistema.
- Tolerância a falhas (*fault tolerance*) – ter robustez contra o atraso de mensagens, a perda ou a replicação ou falha de processos.
- Colaboração entre coletor local–distribuído. Alguns coletores distribuídos consistem de modificações superficiais de coletores de monoprocessadores (um por *host*) combinados com um coletor inter-espaço.
- Desligamento do mutador local. Algumas pesquisas **[Piq96]** sugerem que em ambientes que suportem migração de objetos é muito ruim usar coletores distribuídos que confiam nas mesmas estruturas também usadas pelo mutador (a que encontra o objeto ou a dos ponteiros guia).
- Ciclos distribuídos.

Todas as questões têm que ser consideradas, mas a primeira a considerar acima é particularmente importante desde que a exatidão do algoritmo distribuído dependa dela.

3.3 Coletores de contagem de referência distribuída

Contagem de referência distribuída é uma extensão simples da contagem de referência para monoprocessador **[Col60]**. Aqui o contador é associado a cada objeto, mas é geralmente de modo superficialmente diferente. Cada objeto guarda no seu contador um valor que representa o número de *hosts* (não objetos) que fazem referência a ele. Quando o valor do contador chega a zero, o objeto não é mais referenciado remotamente e o problema é conseqüentemente reduzido no caso monoprocessado. Como no caso não distribuído, o benefício obtido ao se intercalar pequenas etapas de coleta com a computação é preservado.

Um novo problema que se tem para resolver na arquitetura distribuída é o impedimento da reclamação do objeto enquanto houver referências vinculadas a ele. Isto pode acontecer se mensagens forem entregues na ordem diferente da esperada. Por exemplo, se uma mensagem exclui a última referência de um objeto que tomou a cópia da mensagem (a cópia referenciada por outro nó), o objeto será reclamado incorretamente. Um modo de se tratar isto é posteriormente descrito nesta subseção.

Outro problema desta abordagem é o modo de como ela trata mensagens duplicadas: isto resultará em coletas prematuras de objetos. Entretanto, utilizando uma lista de referências isto pode resolvido.

O cenário seguinte mostra como a condição de competição pode ocorrer num algoritmo de coleta de lixo distribuído baseado na contagem de referência:

1. O *host* A envia uma cópia do objeto P (de propriedade do *host* C) para o *host* B;
2. A destrói a referência de P e notifica C;
3. C (o proprietário de P) recebe a mensagem e pensa que a última referência a P foi excluída e, portanto destrói P;
4. B recebe a cópia de P de A e notifica C;

5. C percebe que P foi incorretamente excluído! (ou, mesmo o pior, ele pensará que uma nova referência tenha sido criada para algum outro objeto Q que tomou o lugar de P).

A Figura 3.1 ilustra o cenário descrito acima (R representa o nó raiz em cada máquina).

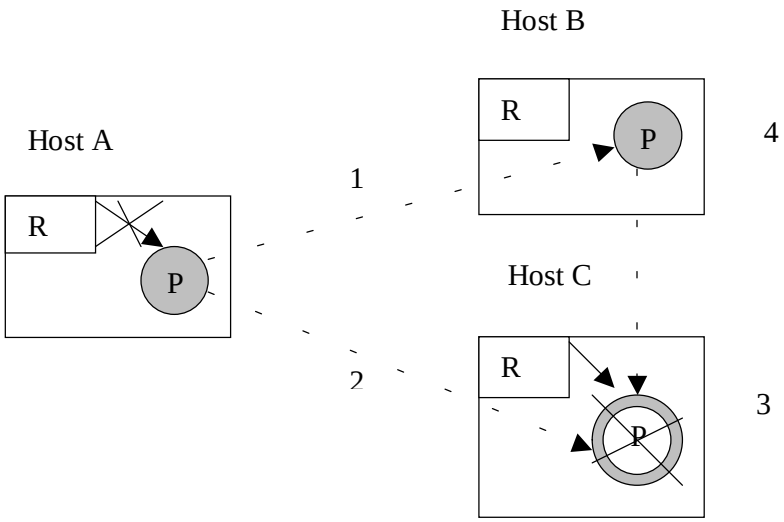


Figura 3.1 - Condição de competição na coleta de lixo distribuído

Problemas similares ocorrerão nos algoritmos *mark-sweep* (baseado no tracejamento), onde há necessidade de sincronização entre a fase *mark* individual e a fase *sweep* distribuída. Algoritmos baseados em contagem de referência, em *mark-sweep* e em outros tipos de algoritmos de coleta de lixo distribuído serão discutidos nas seções seguintes.

Um último comentário ainda pode ser considerado: este tipo de coletor é considerado melhor ajustado para arquiteturas fracamente-acopladas, desde que toda referência de cópia/exclusão envolva mensagem de controle (incremento/decremento) enviada ao proprietário do objeto. Por outro lado, também não está claro que todos os algoritmos de tracejamento têm desempenho melhor neste ponto de vista (uma quantidade de sobrecarga de comunicação), onde o tracejamento é baseado no exame de todos os objetos e conseqüentemente muitas mensagens são emitidas. É difícil dizer qual abordagem escala melhor em geral, e por isso, cada algoritmo precisa ser avaliado separadamente. Outra razão para se fazer esta difícil comparação é que somente alguns algoritmos examinados foram

implementados, e não havia nenhum trabalho completo na tentativa de compará-los **[Sou00]**. Isto é importante, pois cada abordagem é vinculada a um sistema específico.

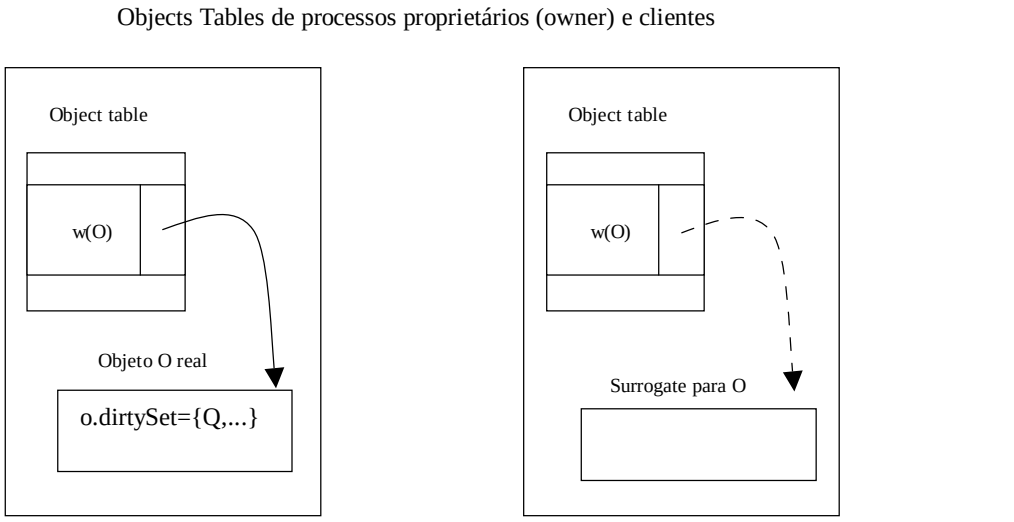
3.4 Coletores em objetos de rede

Birell e sua equipe **[BEM+93]** propõem um algoritmo de coleta por listagem de referência distribuída para suportar programação orientada a objetos distribuída. Os autores apresentam um esboço dos seus métodos, mas não dão detalhes de implementação.

Objetos visíveis a outros nós são chamados objetos de rede. O processo cliente pode armazenar referências de um objeto real através de um objeto hospedeiro (*surrogate*) que se comunica com o proprietário (*owner*) através de chamadas de procedimentos remotos. O proprietário é o *host* que contém o objeto real e também contém a listagem de referências de cada objeto (guardada dentro do objeto em uma estrutura chamada ***dirtySet***).

Esta abordagem não será discutida em detalhes, mas em vez disso, daremos novas idéias e o mecanismo que se utiliza para se evite condição de competição e a tolerância a falha.

Um novo conceito é introduzido: a ***weak reference***. Este é um tipo especial de referência (como uma referência normal) que permite o objeto referido ser elegível para coleta de lixo. Este conceito foi posteriormente introduzido em Java no nível de interface de aplicação a fim de dar algumas interações limitadas com o coletor de lixo local. Um benefício importante é que os coletores de lixo local e global são desacoplados (independentes), e qualquer tipo de coletor local pode ser usado contanto que ele disponibilize mecanismos de *weak reference*. A Figura 3.2 mostra como o objeto hospedeiro é referenciado no cliente e no servidor (**W(o)** representa um identificador único para o objeto **O** em um sistema distribuído).



Fonte:[BEN+93]

Figura 3.2 - Objetos de rede

As possíveis condições de competição são evitadas usando um mecanismo baseado em confirmações. A idéia é pegar uma referência para o objeto (tanto para o *surrogate* quanto para pegar a entrada do *dirty* no *dirtySet*) até que a transmissão da referência seja confirmada. O overhead da mensagem consiste somente numa mensagem extra, desde que objetos de rede sejam transmitidos como resultados/argumentos das invocações de procedimento remoto. Há de se esperar apenas alguma sobrecarga de CPU.

Um bom grau de tolerância a falha é alcançado usando-se mecanismos diversos. Todos os tipos de falhas de mensagens são tratados: mensagens perdidas, mensagens atrasadas (são usados números de seqüência), mensagens duplicadas (isto é uma propriedade intrínseca de algoritmos baseados na listagem de referência). Falha nos processos (ou finalização) é detectada através do envio de mensagens *ping*.

Esta abordagem não dá solução para coleta de ciclos distribuídos.

3.5 Migração de objetos

Uma abordagem interessante que não falha em qualquer categoria acima mencionada é a migração de objeto **[Bis77]**. A idéia básica é muito simples: ao invés de enviar mensagens aos

hosts, tenta-se migrar objetos de modo que a coleta distribuída (coleta de ciclos distribuídos) não seja necessária. Em outras palavras, transformar ciclos distribuídos em ciclos locais que podem ser removidos por qualquer coletor baseado em tracejamento.

A questão chave é como escolher uma boa heurística pra decidir qual objeto é suspeito para ser migrado e ser coletado.

Entretanto, há mais um problema sério com esta abordagem – ele não acomoda bem indireções. Isto significa que é possível que a migração de um objeto pode resultar na criação de novos objetos clientes, complicando os vínculos entre as referências.

Outro aspecto é que os objetos deveriam ser normalmente movidos baseados no critério que carrega o balanceamento da coleta de lixo. O coletor de lixo deveria interferir no processo do mutador o menos possível.

3.6 Tracejamento parcial

[RJ96] propõe um mecanismo para incrementar o modelo de objetos de rede com coleta de ciclos distribuída.

Neste algoritmo, apenas um tracejamento parcial é realizado e as raízes tracejadas são, novamente, escolhidas entre "suspeitos". Nenhum mecanismo razoável para escolha de suspeitos é descrito (os autores mencionam como suspeitos quaisquer objeto que não é localmente referenciado).

O algoritmo opera em três fases:

1. **Mark-red** – identifica o sub-grafo distribuído que pode ser lixo: esforços subseqüentes do tracejamento parcial são confinados para este grafo isolado; neste passo (possivelmente incompleto), os objetos suspeitos do fecho referencial transitivo são marcados com **red** (vermelho); para cada objeto X tracejado, um *RedSet*(X) é incrementado. *RedSet*(X) contém todos os locais que contém referências a X;
2. **Scan** – determina que membros do sub-grafo são realmente lixo; isto é feito através da comparação *ClientSet*(X) com *RedSet*(X) para cada objeto marcado com *red*. O

ClientSet(X) é o mesmo nome usado pelos autores para os X's *dirtySet* (na terminologia Objetos de Redes). Se, como resultado, um objeto é detectado como vivo, então ele é marcado com **green** (verde). No segundo passo, todos os objetos alcançáveis das raízes locais ou dos objetos reais pintados de verdes são agora repintados de verde pelo processo ***local-scan*.** Isto parece ser uma fraqueza desta abordagem – uma rotina de tracejamento local separado pode ser implementado pelo coletor de lixo distribuído. A idéia é que através do coletor local pode-se usar um coletor local baseado em tracejamento e o coletor de lixo distribuído fica sem acesso a ele por causa do modelo de objetos de rede utilizado;

3. **Sweep** – todos os objetos vermelhos são parte de ciclos distribuídos. Suas entradas na Tabela Objeto são removidas e sistema de Objeto de Rede cuidará desta coleta. A fase mark não precisa achar o cerco referencial transitivo completo dos surrogates suspeitos. Por conseguinte, este coletor de lixo distribuído é conservativo no (usual) custo–eficiência (e escalabilidade).

Além do problema mencionado em 2, o algoritmo tem outra fraqueza: ambas as fases *Scan* e *Sweep* precisam usar algoritmos de finalização distribuído para detectar o fim do passo respectivo.

Os autores afirmam que seus algoritmos são tolerantes a falhas como no sistema de Objetos de Rede. Entretanto, não está claro se todo seu sistema de mensagem pode ser inteiramente construído no topo do mecanismo de mensagem oferecida pelos Objetos de Rede.

3.7 Algoritmo de contagem de referência ponderada

A contagem de referência ponderada (*weighted reference counting*) é similar a contagem de referência ordinária para ambiente distribuído, onde é importante a redução do custo da comunicação. Os primeiros trabalhos neste sentindo apareceram nas implementações de Bevan **[Bev87]** e Watson&Watson **[WW87]**. A idéia básica deste algoritmo é trabalhar com o armazenamento de um peso de valor positivo para cada referência e uma associação a um

contador de referências (*Reference Count* - RC) de cada célula. Cada célula passar a ter um invariante: seu peso é igual a soma de todos pesos que fazem referências para ela.

3.7.1 Descrição do algoritmo aplicado a objetos

Quando um objeto é criado, é atribuído um contador de referências arbitrário que pode ser restrito a potência de dois para facilitar a operação de divisão em duas partes. É também atribuída uma referência inicial para o objeto através de um peso (uma referência ponderada) igual ao contador. Quando a referência é duplicada, o peso da referência é dividido resultando em duas novas referências, onde não é necessária nenhuma comunicação com o objeto e nem mudar o contador de referência. Na exclusão de uma referência, uma mensagem para decréscimo do contador de referências, contendo o valor do peso da referência excluída, é enviada ao objeto e ao contador que sofre este decréscimo. Assim, quando o contador de referência do objeto chega a zero, sabe-se que ele poderá ser coletado. Ao fazer isso, deve-se enviar uma mensagem de decréscimo do contador para todos os objetos que mantém estas referências.

Nos algoritmos desenvolvidos na proposta original de **[Bev87]** e **[WW87]**, cada célula ou objeto passa a ter um contador que é um peso de valor inteiro positivo. Um objeto tem campos que armazenam ponteiros. Esses ponteiros fazem referências aos outros objetos e um peso (*weight*) é associado a cada ponteiro. O campo contador de referências do objeto contém o peso total de todos os ponteiros que fazem referência a ele.

Para formalizarmos estes algoritmos definiremos as operações envolvidas para atualizações no grafo de objetos. Mas, antes definiremos as seguintes notações:

- Objeto: R;
- <R,S>: Um ponteiro do objeto R para o objeto S;
- *Weight*(<R,S>) ou W(<R,S>): Peso do ponteiro de R para S;
- RC(S): Contador de referência de um objeto S;

- RC(N) = $\sum_{N} W$ (<X, N>): Representa o invariante de todos os objetos X que apontam para N, ou seja, representa o somatório de todos os pesos parciais dos objetos que aponta para N.

Assumiremos também que o grafo é formado por objetos em uso que se iniciam num ponto denominado raiz. Todos os objetos em uso (ativos) são transitivamente conectados à raiz. Assumiremos também que todos os objetos que não estão ativos estarão vinculados formando um conjunto, a *free-list*. Todo objeto que não for alcançável a partir da raiz será considerado lixo. Os pseudoalgoritmos a seguir serão descritos em termos de procedimentos primitivos no grafo.

3.7.1.1 Algoritmo aplicado a objetos – procedimentos

New(R): Seleciona um objeto U da *free-list* e cria o ponteiro <R,U>, onde R é um objeto transitivamente vinculada à raiz. O contador de referência de U e o peso do ponteiro <R,U> são inicialmente igual ao peso máximo denotado por ***w***.

```
New(R) =
   if (free-list is not empty) then
       select U from free-list
       set RC(U) = w
       make pointer <R,U>
       W(<R,U>)= w
   else
       Msg "No cells available"
```

Algoritmo 3.1 – Criação de um objeto novo através do procedimento *New*

Copy(R,<S,T>): Cria o ponteiro <R,T>, onde R e S são objetos vinculados transitivamente à raiz e o ponteiro <S,T> já existe. O peso de cada ponteiro <R,T> e <S,T> é igual à metade do peso original de <S,T>. Nenhuma comunicação se dá com T, que é o proprietário do objeto, uma vez que ele pode ter sua referência copiada por outros objetos sem que haja necessidade de envio de mensagem de notificação.

```
Copy(R,<S,T>) =
     make pointer <R,T>
     W(<R,T>) = W(<S,T>) / 2
     W(<S,T>) = W(<R,T>)
```

Algoritmo 3.2 – Criação de um objeto novo através do procedimento *Copy*

Delete(<R,S>): Remove o ponteiro <R,S> do grafo e dispara uma ação para reajustá-lo. Somente agora o processo de intercomunicação recomeça. O objeto R enviará para S o peso do ponteiro excluído. O peso é subtraído do contador de referência de S. Se este contador chega a zero, então S está livre e seus filhos podem ser reclamados (desvinculados) recursivamente pelo *Delete*. O objeto T pertence ao conjunto *Sons*(S) se, e somente se, há um ponteiro <S,T>. O objeto é desvinculado do grafo e devolvida à *free-list*.

```
Delete(<R,S>) =                                    (In processor R)
      send Message-Delete <R,S> to S
      remove(<R,S>)

Handle-Delete(<R,S>) =                             (In processor S)
      RC(S) = RC(S) – W(<R,S>)
      if (RC(S) = 0) then
         for T in Sons(S) do
             Delete(<S,T>)
         Link-to-free-list(S)
```

Algoritmo 3.3 – Exclusão de um objeto através do procedimento *Delete*

Pode-se utilizar a representação logarítmica para reduzir o tamanho do campo dos pesos. Quando um objeto é criado, seu contador de referências é igual a maior potência de

dois que o contador de referência pode assumir. Os pesos de todos os ponteiros também são reduzidos a potência de dois. Quando um ponteiro for copiado, seu peso é dividido e re-armazenado em dois novos ponteiros com pesos. Vale observar que isto restringe o peso de todos os ponteiros a representação da potência de dois e que somente podemos dividir por dois as referências ponderadas.

O maior problema com esse esquema é quando um ponteiro com peso igual a 1 é copiado, pois este peso não pode ser mais dividido em dois valores não inteiros. A Figura 3.3 mostra a solução para este problema com a inclusão de um objeto ou uma célula de indireção (CI) no grafo. A figura mostra o grafo no qual o objeto A contém um ponteiro para P que possui peso 1. P possui um contador de referência denotado por ***rc***. Outro objeto B é criado como cópia que aponta para P em A, mas o peso 1 do ponteiro não pode ser mais dividido. Para resolver esta situação um objeto de indireção N é criado e os campos nos objetos A e B são atualizados para apontar para ele (Figura 3.3 b). O novo objeto N aponta para P que está com peso 1, mas como N é criado com o novo valor máximo (128), este pode ser divido dando-se o valor 64 para A e B.

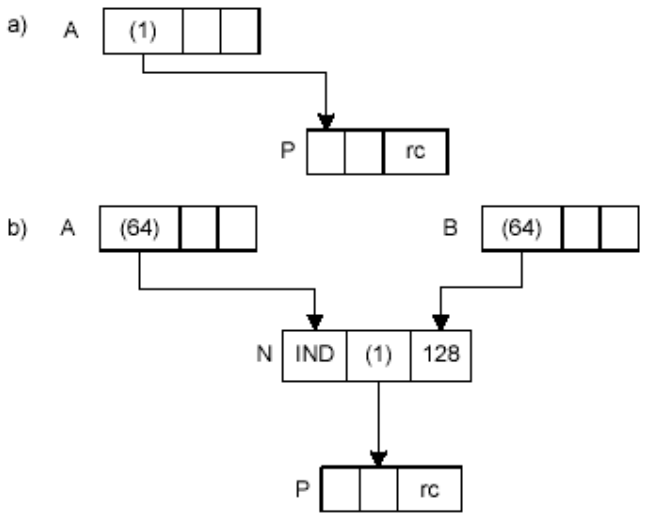


Figura 3.3 - Criação da célula de indireção N

A contagem de referência ponderada tem muitos atributos positivos. Sua principal vantagem é a simplicidade. A implementação de tal esquema envolve cálculo com logaritmos

e potências. Assim, o algoritmo é também verdadeiramente um coletor de lixo real-time com execução em harmonia com a aplicação. A comunicação através na rede é reduzida por ter somente mensagens de decréscimos. E finalmente, ela é relativamente eficiente em termos de espaço.

A desvantagem deste algoritmo é que as células de indireção adicionam uma camada extra para algumas referências remotas e isto envolve comunicação com mais de um *host* através da rede durante a exclusão de referências. Adicionalmente, isto significa que se aquele *host* que contém esta célula fica fora do ar, então aquela referência ao objeto para outro *host* via célula de indireção não pode acessar mais o objeto. Finalmente, a contagem de referência ponderada não é flexível à perda ou a duplicação de mensagens. Se uma mensagem é perdida, então o objeto poderá ser impedido de ser coletado. Se ela é duplicada, um objeto poderá ser coletado prematuramente. Um problema como está sobre a confiabilidade da rede subjacente. Entretanto, se é desejável que o algoritmo seja independente da rede, então devemos avaliar sua computação.

Watson & Watson **[WW87]** implementaram este algoritmo e o avaliaram durante dois anos. O número de células de indireção foi de pelo menos 0.1% dos objetos criados, embora eles não tenham afirmado que proporção de referências foi obtida indiretamente através destas células. Além disso, quando a célula de indireção foi encontrada no grafo, ela foi simplesmente sobrescrito pelo resultado da avaliação. Assim, todas as referências subseqüentes não têm que passar através dessas células. No entanto, esses resultados são específicos ao ambiente em que o algoritmo foi implementado e não se pode tirar conclusões gerais.

Na medição de desempenho do seu algoritmo, Bevan **[Bev87]** levou somente em consideração linguagens de programação funcional. Assim, o número de referências médias que ele encontrou situava-se entre 1,25 e 1,45; com objetos mais usados ele encontrou médios entre 1,6 e 2,3. No entanto, a maioria dos objetos têm somente uma ou duas referências. Entretanto, devemos lembrar que este tipo de avaliação estava especificamente limitada ao ambiente no qual Bevan utilizou seu algoritmo, e com isso, levando ao mesmo problema comentado em Watson & Watson.

3.7.2 Tabela de peso de referência

Uma alternativa para a contagem de referência ponderada foi proposta por Corporaal *et al* em 1990 **[CV+90]** que usava quase o mesmo esquema. A única diferença entre os métodos é como o algoritmo trata a referência ponderada quando ela chega a um. Ao invés de criar uma célula de indireção, Corporaal utiliza uma tabela para armazenar o novo contador de referência. A referência ainda aponta para o objeto, além de conter um ponteiro para uma entrada de tabela. Quando uma referência encontra este estado, ela designa um peso de empréstimo (*Borrowed Weigh* – BW) ao invés de uma referência ponderada. Isto significa que se aquela referência é copiada, tanto a referência resultante compartilha a BW quanto a referência que pegou a entrada da tabela. Assim, quando uma referência é excluída, o contador de referência da tabela deverá sofrer um decréscimo através da BW daquela referência. Corporaal afirma que cada espaço de endereço tem sua própria tabela de peso e dá sugestão de que a entrada vá para dentro da tabela na localização da referência copiada.

O método de Corporaal elimina o problema das camadas de indireção adicionais encontradas na contagem de referência ponderada. De fato, não há problemas quando muitas referências são copiadas, uma vez que não é necessário criar CIs; agora simplesmente empresta-se mais pesos. Como não há mais CIs, cada desreferenciamento de um objeto é feita de forma direta e não existe mais o problema do host que contém o CI saia do ar, o que evita que um objeto em outro *host* seja desreferenciado.

3.7.3 Algoritmo de contagem de referência ponderada distribuída com pesos totais e parciais

Um algoritmo foi proposto por Dickman **[Dic92]** para resolver os problemas das células de indireção e prover um ambiente mais tolerante a falhas onde um objeto não lixo não seja coletado. Como resultado, o algoritmo é intrinsecamente mais conservativo que os demais mencionados anteriormente. Como cada objeto é associado a um peso total (*total-weight*) de tamanho de uma palavra. Se este valor é zero, ele é chamado de peso-total infinito. Objetos que contém um peso total infinito não podem ser coletados. Um peso parcial (*part-weight*) é

associado a cada referência. Este peso consiste de dois campos: a mantissa e um expoente. Se estes campos são zero, então cada um é chamado de peso parcial nulo. Ao receber uma mensagem cuja referência com um peso parcial nulo foi excluído, o peso total associado com um objeto é assinalado com um peso total infinito.

Para a implementação dos algoritmos, isto significa que se a referência com peso parcial 2 é copiado, a cópia resultante e a original são assinaladas com um peso parcial nulo. Como isto implica que o peso parcial de valor dois vai desaparecer, então uma posição de entrada de tabela apropriada não pode ser desalocada e o objeto associado não pode ser coletado. Esta implementação do algoritmo envolve cada espaço de endereço mantenha uma entrada de tabela de posições indexadas. Cada posição de entrada da tabela pode conter um peso parcial e um ponteiro para o objeto; pode ser parte de uma lista ligada de posições livres; ou pode indicar que um objeto foi migrado e assim conter a nova localização do objeto.

3.7.4 Vantagens e desvantagens do algoritmo de contagem de referência ponderada

3.7.4.1 Vantagens da contagem de referência ponderada:

1. Elimina as mensagens de comando de incrementos e decrementos;
2. Elimina condições de competição;
3. A entrega de mensagens fora de ordem não é mais problema.

3.7.4.2 Desvantagens da Contagem de Referência Ponderada:

1. Limitado número de duplicações. Mas há possíveis soluções:
   1.1. Quando o peso parcial não puder ser dividido, o emissor faz uma requisição ao proprietário que adicione novo valor ao peso total, os quais assegura ao emissor incrementar seu peso parcial através desse valor;
   1.2. Cria um item de entrada indireto no espaço do emissor. Todas as referências subseqüentes se referem a este item e não mais diretamente ao espaço proprietário;
2. Não é flexível à perda de mensagens: O peso total torna-se maior que a soma dos pesos parciais. Desse modo o objeto nunca será coletado;

3. Não é robusto à duplicação de mensagens: O peso total torna-se menor que a soma dos pesos parciais. O objeto pode ser prematuramente coletado;
4. Não consegue coletar lixos cíclicos distribuídos.

Pode-se evitar o problema da duplicação de mensagens através do “relaxamento” na verificação dos pesos, onde o peso total pode ser maior ou igual ao somatório dos pesos parciais.

3.7.5 Cenários do algoritmo de contagem de referência ponderada distribuída

Os sistemas distribuídos são particionados em espaços de memórias distintas. Estes espaços interagem através de passagens de mensagens. Assim, os objetos conseguem enviar suas referências entre si. A comunicação entre os espaços não é confiável. Mensagens podem se perder, duplicar, atrasar ou chegar fora de ordem. Um espaço pode falhar, seja por problemas de hardware ou software, por reinicializações etc. Quando ocorre uma falha, assume-se que não houve envio de mensagens.

A Figura 3.4 mostra os itens que serão utilizados nas figuras seguintes no cenário de criação e duplicação.

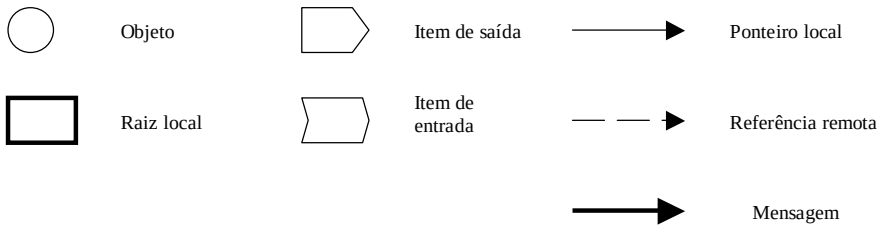


Figura 3.4 – Itens do cenário de objeto distribuído

Para ilustrar nosso cenário de criação, duplicação e exclusão, utilizaremos as seqüências dessas ações com três *hosts* A, B e C. O *host* B é o proprietário do objeto que vai ser

referenciado por dois outros objetos remotos nos *hosts* A e C. Admitiremos que o valor máximo do peso que o objeto ***v*** de B pode assumir é 64.

A Figura 3.5 ilustra a criação do objeto ***x*** de A que vai fazer uma referência ao objeto ***v*** de B. Então B cria uma referência para ***v*** e a envia para A através de uma mensagem que transporta o peso parcial calculado, cujo valor é metade do peso total, ou seja, 32.

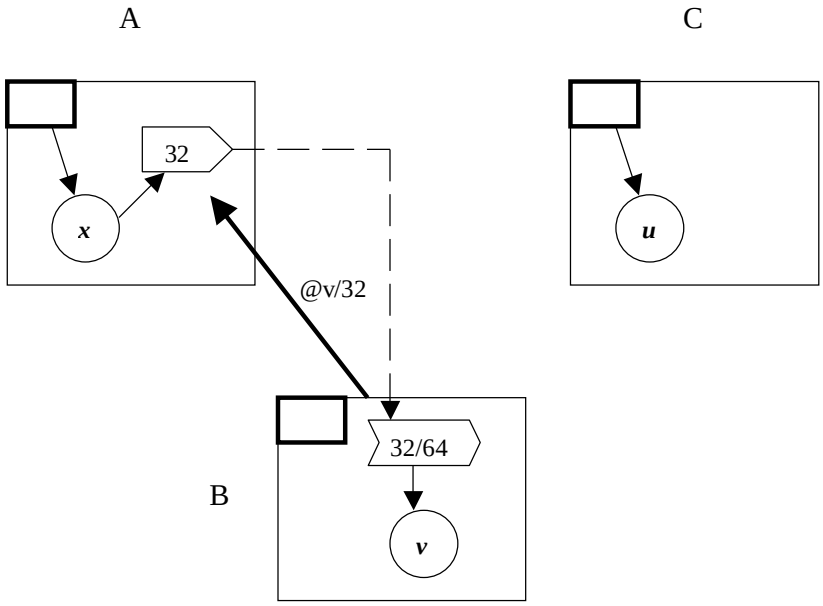


Figura 3.5 – B cria uma referência para ***v*** e a envia para A

A Figura 3.6 ilustra a criação do objeto ***u*** de C que vai fazer uma referência ao objeto ***v*** de B copiando a referência contida em A. Então A duplica a referência para ***v*** e a envia para C através de uma mensagem que transporta o valor do peso parcial, cujo valor é metade desse valor, ou seja, 16.

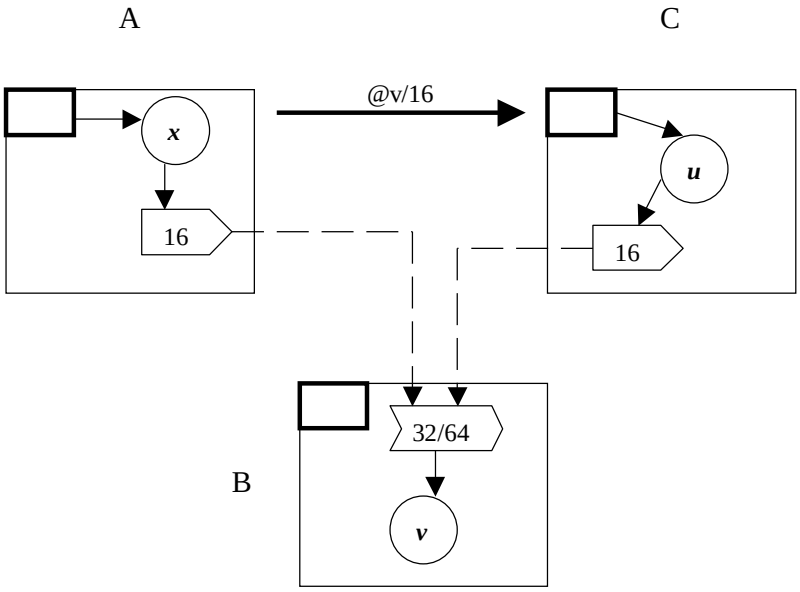


Figura 3.6 – A duplica referência para ***v*** e a envia para C

A Figura 3.7 ilustra a exclusão da referência que C faz ao objeto ***v*** de B. C elimina sua referência e envia uma mensagem para A de modo que seu peso sofra um decréscimo do valor peso enviado. O peso total do objeto ***v*** de A passa a ser agora 48.

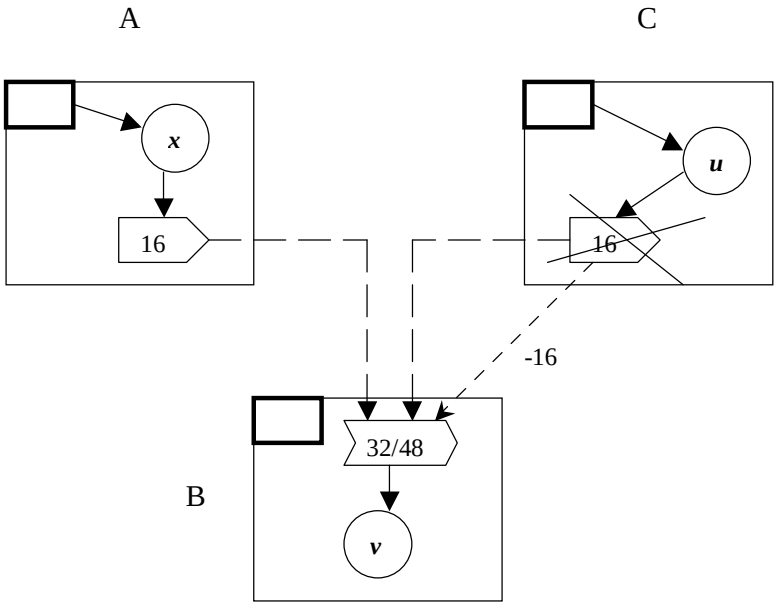


Figura 3.7 – C exclui sua referência para o objeto de B

A exclusão da última referência ao objeto ***v*** de B é ilustrada na Figura 3.8. A elimina sua referência e envia uma mensagem para A de modo que seu peso sofra um decréscimo do valor peso enviado. Os pesos do objeto ***v*** de A passam agora a ter o mesmo valor, ou seja, 32. Com isso, o coletor pode reclamar objeto de B que não é mais referenciado por mais nenhum objeto remoto.

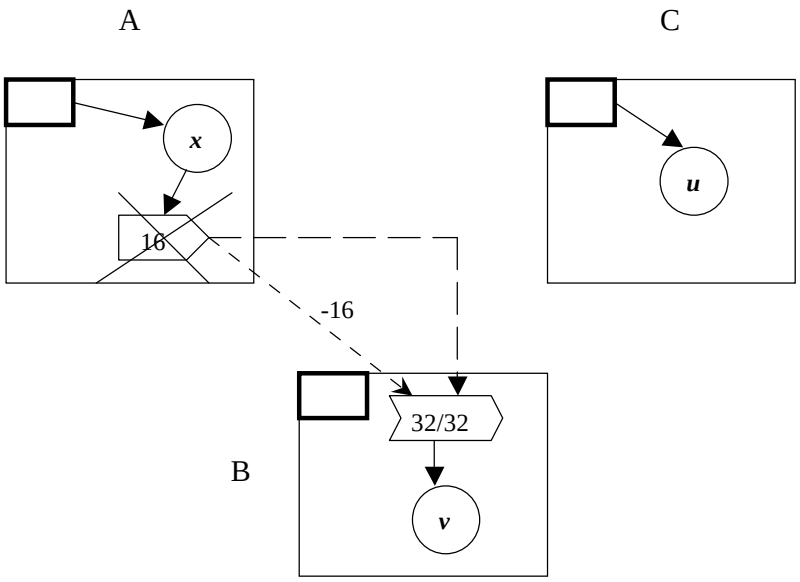


Figura 3.8 – A exclui sua referência para o objeto de B

Capítulo 4

Um algoritmo para gerência de páginas Web baseado na contagem de referência ponderada

Na World Wide Web várias páginas “apontam” para outras páginas no mesmo servidor ou em servidores remotos. Estas páginas são ligadas entre si através dos hiperlinks baseados na linguagem HTML suportada pelos navegadores. Um dos problemas que ocorrem atualmente é que não existe nenhum controle sobre consistências dessas ligações. Podemos incluir um hiperlink numa página para fazer referência à outra página, mas nada nos garante que esta estará sempre ativa. Este cenário também não se limita ao ambiente da Web. Podemos citar um cenário em um ambiente de desenvolvimento em uma empresa no qual vários programadores e analistas estão em construindo e mantendo páginas HTML em paralelo. Os softwares que utilizam para escrever estas páginas não provêem nenhum mecanismo que garanta uma correta consistência de vínculos entre elas. Um programador pode querer excluir uma página, mas não tem certeza se a mesma está sendo referenciada por outra página. Geralmente os programadores criam diretórios para conter várias páginas relacionadas a uma parte do sistema de acordo com sua classificação de atividade, mas o software simplesmente não garante a consistência dinâmica dos vínculos entre elas. Softwares tipo *FrontPage* da Microsoft **[MSFP]** ou o *Dreamweaver da Macromedia* **[Dre]** são praticamente meros editores de texto HTML. Em ferramentas CASE (*Computer Aided Software Engineeing* – Engenharia de Software Apoiada por Computador) voltadas para desenvolvimento completo de aplicações Web, guardar os vínculos entre várias páginas seria algo imprescindível. Podemos citar a *WebRatio* **[WR]** com exemplo de uma dessas ferramentas CASE que é baseada na linguagem de modelagem para Web definida por *WebML* **[WML]**.

A proposta pioneira desta dissertação é aplicar o algoritmo de contagem de referência ponderada de Lins **[Lin02b]** em ambiente distribuído para o gerenciamento consistente de páginas Web.

4.1 O algoritmo de contagem de referência ponderada e os hiperlinks

A proposta que iremos apresentar para dar consistência às ligações entre as páginas HTML é baseada no algoritmo de contagem de referência ponderada que é utilizado na coleta de lixo distribuído. Este algoritmo foi originalmente projetado para gerenciar a coleta de lixo em um ambiente de objetos distribuídos.

Diferentemente das estruturas dinâmicas controladas por programas que executam sobre um sistema operacional, as páginas HTML são textuais e não possuem uma estrutura dinâmica em si. Somente os hiperlinks asseguram as ligações entre as páginas e estes são "inseridos" pelo programador de forma não automática. Não existe nenhum mecanismo que indique que uma determinada página HTML esteja sendo referenciada por outra página. Ou seja, não há como saber se alguma página tem um hiperlink em outra página que aponte para ela. Na Figura 4.1, a PagC não tem nenhum indicador de que esteja sendo referenciada por PagA e PagB.

O problema reside no fato de se querer retirar uma página de um diretório sem saber se ela está sendo apontada por outra. Para sabermos isto, teríamos que varrer toda a World Wide Web em busca das outras páginas para encontrar algum hiperlink que aponte para ela. Face às dimensões e interconectividade da Web, tal solução é inviável. A idéia que advogamos nesta dissertação é bem diferente desta e adota um contador de referência que estaria contido na própria página HTML. Fica evidente que essa proposta define uma nova definição sintática no HTML, assim como são XML e SML. A diferença dessa nova definição é que ela não é apenas estrutural e sim interativa. As páginas HTML passariam a interagir de forma autônoma entre si, de forma a atualizar dinamicamente este contador e algum indicador de controle das cores. Estas seriam variáveis da mesma forma que as variáveis de uma linguagem s*cripts* como em JavaScript e VBScript. Quando uma página fosse definida e um determinado

vínculo que apontasse para outra página fosse inserido pelo programador, a página que passou a ser referenciada só tomaria conhecimento que uma outra passou a referenciá-la após um evento de ativação da nova página. Este evento pode ser uma ação de colocar a nova página no "ar" ou o primeiro clique de ativação do vínculo para outra página referenciada pelo hiperlink.

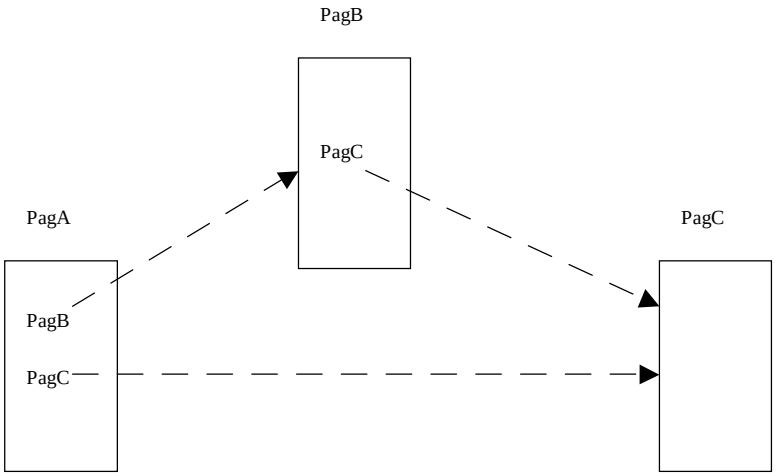


Figura 4.1 - A PagA tem dois hiperlinks: Um para PagB e outro para PagC. A PagB tem apenas um hiperlink para PagC

4.2 Um novo ambiente para o algoritmo de contagem de referência ponderada aplicado a páginas Web

Para aplicar estes algoritmos no gerenciamento consistente de vínculos de páginas Web, definiu-se um modelo de uma nova extensão na linguagem HTML e um novo ambiente para seu gerenciamento.

Assim como foi definido um arquivo separado para definição de estilo de fontes na W3C **[W3C-b]**, podemos adotar o mesmo esquema em nosso modelo. Nessa definição de estilos de fontes, pode ser criado um arquivo externo com a extensão **".css"** que é relacionado com o documento em HTML conforme ilustra a Figura 4.2 a seguir.

```
<html>
<head>
<link rel="stylesheet" href="MeusEstilos.css" type="text/css">
...
</head>
</html>
```

Figura 4.2 - Sintaxe de chamada à folha de estilos de fontes num arquivo externo

Definiremos dois novos identificadores que denotaremos com a seguinte sintaxe: *wt=256 wp=256*. O identificador *wt* indicará o valor do peso total e o identificador *wp* indicará o valor do peso parcial.

Observe-se que sempre adotaremos os valores numéricos para os pesos como potências de dois para facilitar a divisão matemática.

Então, uma página Web passaria a conter um link obrigatório para controle interno, conforme ilustra a Figura 4.3.

```
<link rel="wrc" href="PageName.wrc" type="internal control">
```

Figura 4.3 - Sintaxe de chamada ao controle do contador de referência ponderada

Pela nossa definição do modelo, esse arquivo teria o mesmo nome da página Web, mas com a extensão do nome do arquivo com **“.wrc”** (*weighted reference count*), indicando que é de controle interno. E a página de controle interno fica definida conforme ilustra a Figura 4.4. A cor que indica o estado da página também usa o mesmo mecanismo: *PageColour.wrc*.

```
wt=256 wp=256
```

```
colour
```

Figura 4.4 - Arquivo “PageName.wrc” - Sintaxe dos dados contidos no arquivo para controle interno do contador de referência ponderada. O Arquivo de cor pode ser representado também separadamente.

4.2.1 Cenários dos algoritmos de contagem de referência ponderada

A Figura 4.5 ilustra o cenário quando página B faz apenas uma referência para a página A e a Figura 4.6 ilustra o cenário quando página C copia endereço de A contido em B. C agora aponta para A.

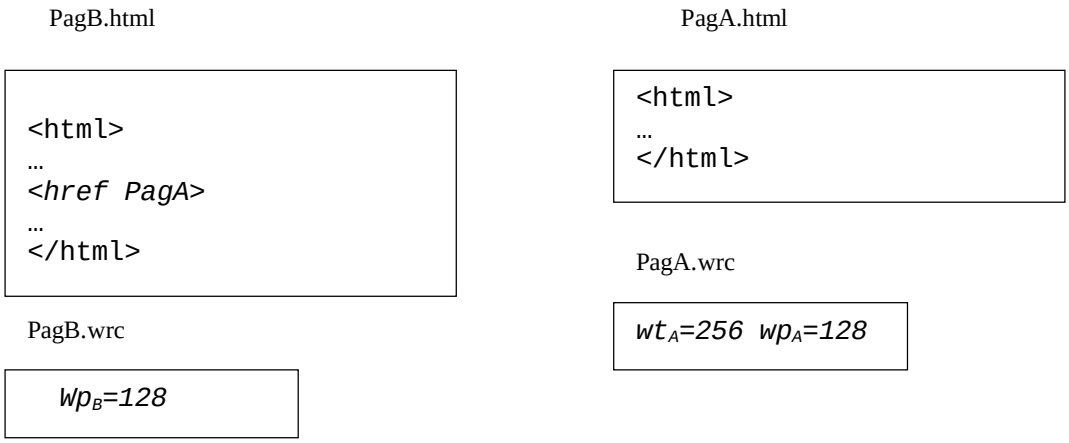


Figura 4.5 – PagB com um hiperlink para PagA

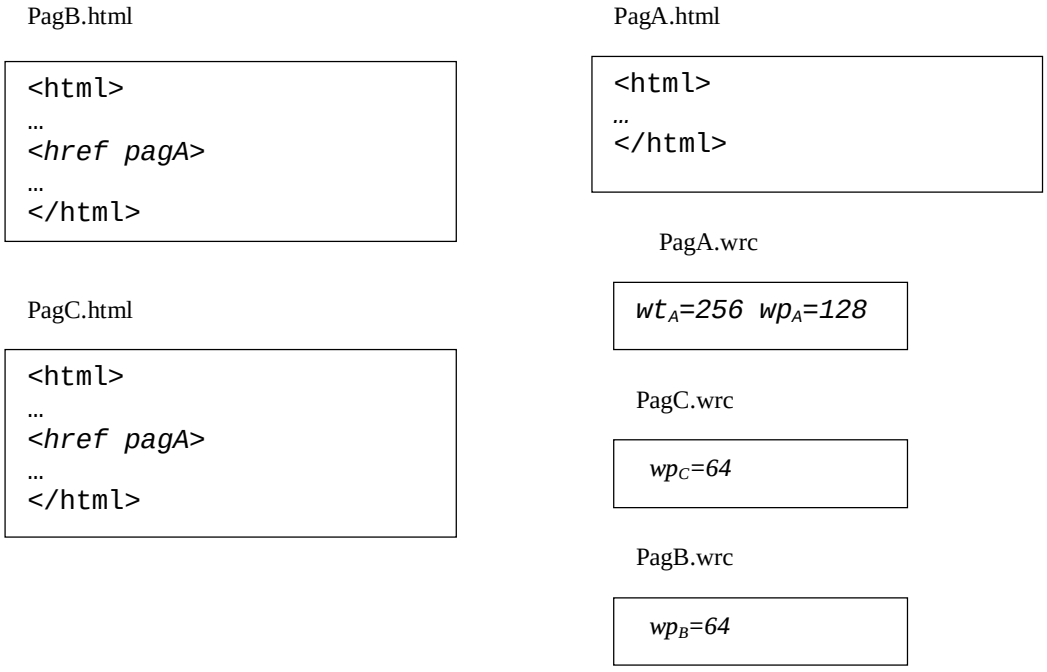


Figura 4.6 – PagB e PagC com hiperlink para PagA

Para evitarmos o problema de esgotarmos mais rápido o contador de referência devido às sucessivas divisões podemos adotar o valor do logaritmo de base dois em vez do valor inteiro, que passaria a ser o valor do antilogaritmo. Agora antes da operação de divisão o antilogaritmo seria calculado.

Por exemplo, $\log_2 256 = 8$, pois $2^8=256$. Onde na PagA ficaria como ilustrado na Figura 4.7.

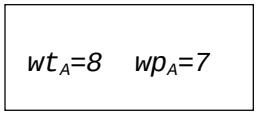


Figura 4.7 – Peso total wp=8 e peso parcial wp=7

Com isso, poderíamos ter $\log_2 65536 = 16$, pois $2^{16}=65536$.

Caso o contador de referência chegue ao valor 1, impossibilitando a divisão, podemos adotar o mesmo procedimento usado na referência de objetos: criar uma célula ou um hiperlink de referência indireta.

Esta célula de indireção na verdade seria um novo arquivo de controle interno de indireção. Este arquivo terá o mesmo nome da página, mas com a extensão de nome do arquivo **".wic"** (*weighted indirection cell*) como ilustra a Figura 4.8 com o novo peso total.

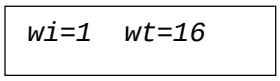


Figura 4.8 – Página de indireção com novo peso total wt=16

Então, teremos um novo indicador na página de indireção: *wi=1*.

Para esse algoritmo se tornar funcional na Web, uma espécie de robô residente nos servidores teria que ser o agente atualizador dos valores dos pesos. Seria também necessário um indicador de controle (um *flag*, por exemplo) que indicasse que uma determinada página que referencia outra não atualize o valor do peso novamente. Somente uma nova página criada que apontasse para outra página poderia desencadear essa atualização. O novo cenário ficaria como ilustra a Figura 4.9.

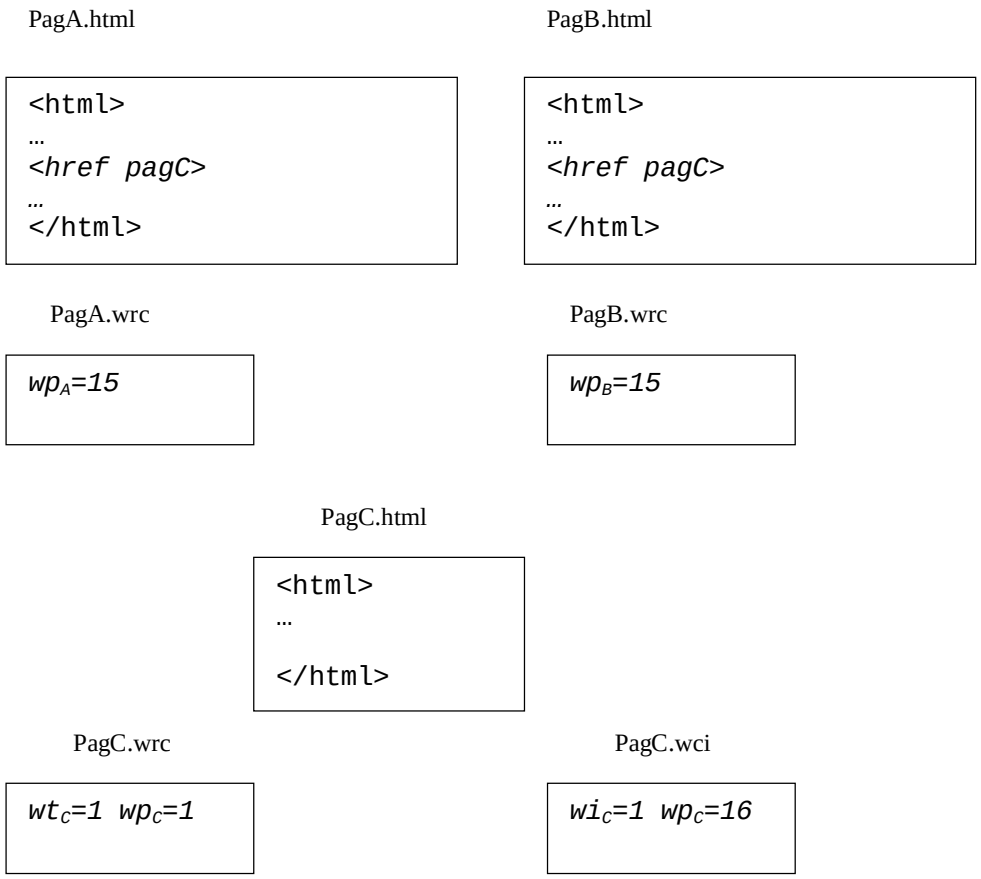


Figura 4.9 – Hiperlinks com páginas de indireção

Para termos uma idéia melhor da evolução destes algoritmos, primeiro apresentaremos os algoritmos desenvolvidos na proposta original de **[Bev87]** e **[WW87]** vistos na seção anterior. Nestes algoritmos, em nossa adaptação para páginas Web, cada página passa a ter um contador que é um peso com valor inteiro positivo. Como uma página pode fazer referência a outras páginas através dos hiperlinks, estes terão também um peso associado. O contador de referência da página conterá o somatório de todos os pesos (peso total) de todos os hiperlinks que apontam para essa página.

Para formalizarmos estes algoritmos definiremos as operações envolvidas para atualizações das páginas Web. Mas, antes definiremos as seguintes notações:

- Página Web: P.
- Ligação entre duas páginas: <P,Q>, onde indica que P possui um hiperlink para Q;
- Peso do hiperlink: representado por W(<P,Q>);
- Contador de Referência da página P: RC(P);
- RC(X) = $\sum_{X} W$ (<P, X>) Representa o invariante de todas as páginas P que apontam para X, ou seja, representa o somatório de todos os pesos parciais das páginas que aponta para X.

Também definiremos o seguinte cenário com servidores, como mostra a Figura 4.10.

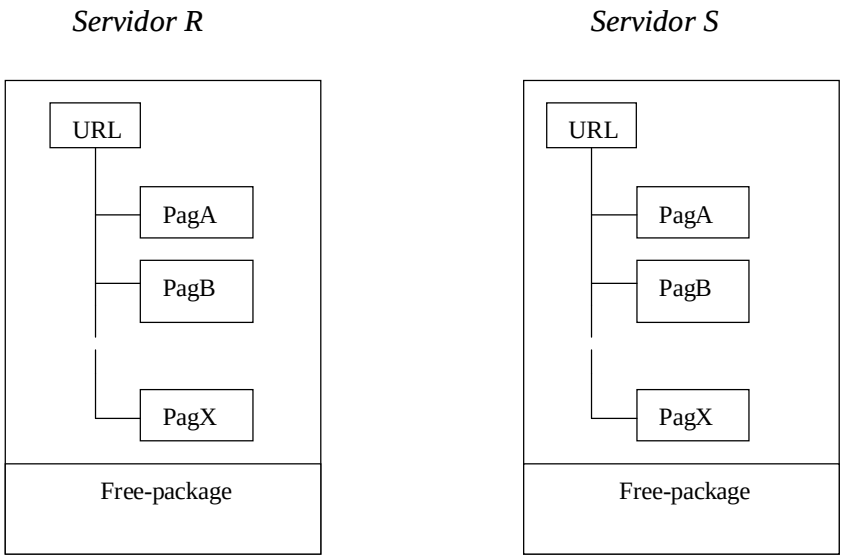


Figura 4.10 - Cenário com dois servidores

O URL é equivalente a raiz do grafo no algoritmo de coleta de lixo. O *free-package* seria o diretório do servidor onde as páginas novas são incluídas pelos programadores e onde as páginas não mais vinculadas ao URL que foram transferidas pelo algoritmo de coleta.

4.2.2 Descrição dos algoritmos

A seguir, mostraremos os algoritmos em pseudocódigo das operações de criação (*New*), de cópia (*Copy*) e de exclusão (*Delete*) sobre uma página Web.

New(P): Caracteriza que uma página Q foi transferida do *free-package* e nela foi incluído o hiperlink <P,Q>, onde P é uma página transitivamente vinculada ao URL de um servidor. Adotaremos ***w*** o valor de um peso máximo para o contador de referência. Tanto o contador de referência de Q quanto o peso do hiperlink <P,Q> terão o peso igual a ***w***.

```
New(P) =
    Q selected from free-package
    RC(Q) = w
    make hiperlink <P,Q>
    W(<P,Q>) = w
```

Algoritmo 4.1 – Criação de uma página através do procedimento *New*

No contexto da programação Web, o procedimento **New** tem um significado apenas conceitual. Em termos práticos, significa que uma nova página (ou uma página modificada), que vai ser colocada "no ar", é atualizada com um novo link para uma página vinculada a um URL já existente.

Copy(R,<S,T>): Caracteriza a inclusão do hiperlink <R,T>, onde R e S são páginas vinculadas ao URL do servidor e o hiperlink <S,T> já existe. O peso de cada hiperlink <R,T> e <S,T> é igual à metade do peso original de <S,T>. Nenhuma comunicação se dá com T, que é proprietário da página.

```
Copy(R,<S,T>) =
       make hiperlink <R,T>
       W(<R,T>) = W(<S,T>) / 2
       W(<S,T>) = W(<R,T>)
```

Algoritmo 4.2 – Cópia de uma página através do procedimento *Copy*

Novamente, no contexto Web, pode-se explicar este procedimento com uma analogia ao proprietário de um telefone: seu número é copiado por terceiros sem que ele tome conhecimento. Ou seja, um hiperlink pode ser copiado (inserido) em uma outra página sem que o proprietário tome conhecimento.

Delete(<R,S>): Caracteriza a remoção do hiperlink <R,S> do grafo do URL e dispara (ou solicita) uma ação para reorganizá-lo. Somente depois o processo de intercomunicação recomeça. A página R enviará para S o peso do hiperlink excluído. O peso é subtraído do contador de referência de S. Se este contador chega a zero, então S está livre e seus filhos podem ser reclamados (desvinculados) recursivamente pelo *Delete*. A página T pertence ao conjunto *Sons*(S) se, e somente se, há um hiperlink <S,T>. Esta operação não exclui as páginas fisicamente, simplesmente as coloca em um pacote (diretório) desvinculado do URL do servidor.

```
Delete(<R,S>) =                                          (In Server R)
       send Message-Delete <R,S> to S
       remove(<R,S>)

Handle-Delete(<R,S>) =                                    (In Server S)
       RC(S) = RC(S) – W(<R,S>)
       if (RC(S) = 0) then
       for T in Sons(S) do
           Delete(<S,T>)
       Transfer S to free-package
```

Algoritmo 4.3 – Exclusão de uma página através do procedimento *Delete*

Como já descrevemos anteriormente, para melhor descrição deste algoritmo, pode-se utilizar pesos que são potências de índice dois. Isto permite uma técnica prática para implementação, pois cada ponteiro passa a armazenar o logaritmo do seu peso. Nas subtrações, deve-se então calcular antilogaritmo do peso primeiro para depois efetuá-las. Células de indireção são utilizadas quando os ponteiros de cópia de peso chegam a uma unidade. Para executar **Copy**(R,<S,T>) quando W(<S,T>)=1, uma página de indireção U é criada. Esta célula de indireção contém um ponteiro para T – o peso do ponteiro é uma unidade e não precisa ser mais armazenado. Tanto R como S para agora a fazer referência à página de indireção, onde cada ponteiro tem peso W/2. Observe que o contador de referência de T não precisa ser alterado e nenhuma comunicação foi necessária.

4.3 O Algoritmo de Lins para contagem de referência ponderada distribuída cíclica

O algoritmo apresentado na seção anterior tem a desvantagem de não conseguir identificar os hiperlinks que formam um ciclo de ligações entre as páginas que não estão mais transitivamente vinculadas ao URL do servidor (vide Figura 1.4 na página 24).

Baseado no algoritmo de contagem de referência cíclica de Hughes **[Hug85]**, Lester **[Les92]** propôs uma extensão do protocolo de contagem de referência ponderada para tratar de estruturas cíclicas em aplicações com referências indiretas, tais como linguagens funcionais. Jones e Lins **[JL93]** apresentaram algoritmos gerais para contagem de referência ponderada para estruturas cíclicas. Lins **[Lin02b]** propôs novo algoritmo de contagem de referência ponderada, baseado no seu algoritmo anterior **[Lin02a]**, para tratar de estruturas cíclicas em ambientes distribuídos. Neste novo algoritmo, além do peso, são acrescentados mais dois campos extras. O primeiro contém uma cor para a célula. O segundo campo é um peso secundário.

Em nossa aplicação do novo algoritmo de Lins, a condição inicial para todas as páginas é que as mesmas ainda não estão vinculadas ao URL do servidor. Neste caso, pode-se imaginar uma situação em que vários programadores estão desenvolvendo suas páginas individualmente mais ainda não as disponibilizaram no servidor.

O procedimento **New(P)** tem a mesma função que a contagem de referência ponderada, mas utiliza a cor verde (*green*) para uma nova página.

```
New(P) =
       Q selected from free-package
       RC(Q) = w
       colour(Q) = green
       make hiperlink <P,Q>
       W(<P,Q>) = w
```

Algoritmo 4.4 – Criação de uma página através do procedimento *New* – com ciclo

Copy(R,<S,T>): Caracteriza a inclusão do hiperlink <R,T>, onde R e S são páginas transitivamente conectadas ao URL do servidor e o hiperlink <S,T> já existe. O peso de cada hiperlink <R,T> e <S,T> é igual à metade do peso original de <S,T>. Assim como no algoritmo não cíclico, nenhuma comunicação ocorre com T, que é o proprietário da página.

```
Copy(R,<S,T>) =
     make hiperlink <R,T>
     W(<R,T>) = W(<S,T>) / 2
     W(<S,T>) = W(<R,T>)
```

Algoritmo 4.5 – Cópia de uma página através do procedimento *Copy* – com ciclo

Assim como no algoritmo de contagem de referência, o *Delete* força uma comunicação. A idéia geral do algoritmo cíclico é executar um *mark-scan* local onde o hiperlink de uma estrutura compartilhada é excluído. Para evitar problemas de sincronismo de comunicação (condição de competição), todos os servidores devem suspender suas atividades durante a execução do *mark-scan* local. Esta condição pode ser "relaxada" adotando a seguinte estratégia. O algoritmo vai trabalhar em duas fases. Na primeira, o grafo exclui o hiperlink marcado, recalcula os contadores das referências internas e possíveis pontos de conexão para o URL e os armazena numa estrutura de dados chamada ***Jump-stack***. Na fase dois, as páginas que estão nessa estrutura são visitadas diretamente e referências externas podem ser encontradas no sub-grafo que passam a ser marcadas como páginas ordinárias (*green*) e terão seus contadores reinicializados. Todos os outros nós serão páginas não vinculas ao URL, podendo ser então transferidas para outro diretório ou pacote fora do alcance do URL.

Delete(<R,S>): Estende o algoritmo de contagem de referência ponderada através da chamada ao *mark-scan* local no sub-grafo se S tiver outras referências.

```
Delete(<R,S>) =                                          (In Server R)
     send Message-Delete <R,S> to S
     remove(<R,S>)

Handle-Delete(<R,S>) =                                   (In Server S)
     RC(S) = RC(S) – W(<R,S>)
     if (RC(S) = 0) then
         for T in Sons(S) do
             Delete(<S,T>)
             Transfer-to-free-package(S)
     else
         broadcast Suspend
         Mark-red(S)
         Scan(S)
         Collect(S)
         broadcast Continue
```

Algoritmo 4.6 – Exclusão de uma página através do procedimento *Delete* – com ciclo

Mark-red(S) é uma função auxiliar que pinta todas as páginas de um sub-grafo S de vermelho (*red*). Esta cor indica que a página pode vir a ser coletada. Cada célula visitada tem seu contador de referência decrementado, liberando somente os pesos que se têm vínculos externos ao sub-grafo.

```
Mark-red(S) =                                            (In Server S)
    if (colour(S) == green) then
       colour(S) = red
       for T in Sons(S) do
           send Message-mark-red(<S,T>) to T

Handle-mark-red(<S,T>) =                                 (In Server T)
    RC(T) = RC(T) – W(<S,T>)
    if (RC(T)>0 && T not in Jump-stack) then
        Jump-stack = T
        Mark-red(T)
```

Algoritmo 4.7 – Fase *Mark-red*

Scan(S) processo de busca que verifica se a *Jump-stack* está vazia. Se estiver, o algoritmo envia as páginas visadas de S para o *free-package*. Se não estiver, então existem nós do grafo ainda a serem analisados. Se seus contadores de referências são maiores que

zero, há vínculos externos a página sobre o URL e os contadores serão restaurados a partir daquele ponto através da chamada ao procedimento *Scan-green*(T).

```
Scan(S) =                                                    (In Server S)
    if (colour(S) == green && RC(S) > 0) then
        Scan-green(T)
    else
        for T in Jump-stack do
            send Message-scan(T)

Handle-scan(T) =                                             (In Server T)
    Scan(T)
```

Algoritmo 4.8 – Fase *Scan*

O **Scan-green(S)** pinta de verde todos os sub-grafo de S.

```
Scan-green(S)=                                               (In Server S)
    colour(S) = green
    for T in Sons(S) do
        send Message-scan-green(<S,T>) to T

Handle-scan-green(<S,T>) =                                   (In Server T)
    RC(T) = RC(T) – W(<S,T>)
     if (colour(T) != green) then
        Scan-green(T)
```

Algoritmo 4.9 – Fase *Scan-green*

O **Collect(S)** recupera todas as páginas do sub-grafo de S (que são lixo) e os transfere para o free-package.

```
Collect(S) =                                                 (In Server S)
    if (colour(S) == red) then
        for T in Sons(S) do
            send Message-collect(T) to T
            Transfer-to-free-package(S)
```

Algoritmo 4.10 – Fase *Collect*

4.4 O algoritmo *lazy mark-scan* de contagem de referência ponderada distribuída

Do mesmo modo que o algoritmo de Lins foi otimizado para melhorar seu outro algoritmo, postergando o *mark-scan* o máximo possível, ele pode ser feito de forma *lazy*. Este novo algoritmo evita executar de imediato o *mark-scan* local toda vez que um hiperlink para uma página é excluído e passa a utilizar uma nova estrutura auxiliar chamada ***Control-queue*** (tem a mesma função do *Control-set*). Na forma *lazy*, o algoritmo pinta de preto (*black*) toda página com múltiplas referências quando ela for excluída. Esta cor indica que ela passa a ser candidata para reciclagem e é colocada na *Control-queue.*

New(P): Caracteriza que uma página Q que foi transferida do *free-package* e nela foi incluído o hiperlink <P,Q>, onde Q é uma página transitivamente vinculada ao URL do servidor. O contador de referência de Q e o peso do hiperlink <P,Q> passam a ter um peso máximo denotado por ***w***. Q é pintado de verde.

```
New(P) =
    Q selected from free-package
    RC(Q) = w
    colour(Q) = green
    make pointer <P,Q>
    W(<P,Q>) = w
```

Algoritmo 4.11 – Criação de uma página com o procedimento *New - Lazy*

Copy(R,<S,T>): não sofreu mudança.

```
Copy(R,<S,T>) =
     make pointer <R,T>
     W(<R,T>) = W(<S,T>) / 2
     W(<S,T>) = W(<R,T>)
```

Algoritmo 4.12 – Cópia de uma página com o procedimento *Copy - Lazy*

Delete(<R,S>): Remove um hiperlink <R,S> do grafo e o re-ajusta. Quando o processo de intercomunicação recomeça, a página R enviará o peso do hiperlink excluído para S. Este peso é subtraído do contador de referência de S e este é testado para ver se está livre. Se este contador chega a zero, então S está livre e seus filhos podem ser reclamados pelas chamadas recursivas do *Delete*. De outro modo, o objeto S é pintado de preto (*black*) e colocado na *Control-queue* (a menos que já esteja lá).

```
Delete(<R,S>) =                                          (In Server R)
      send Message-Delete <R,S> to S
      remove(<R,S>)

Handle-Delete(<R,S>) =                                   (In Server S)
      RC(S) = RC(S) – W(<R,S>)
      if (RC(S) = 0) then
          for T in Sons(S) do
              send Message-Delete <S,T> to T
              remove <S,T>
          Transfer-to-free-package(S)
      else
         if (colour(S) == green) then
             colour(S) = black
         if (Control-queue is full) then
             Scan-queue()
         Back-of-Control-queue = S
```

Algoritmo 4.13 – Exclusão de uma página com o procedimento *Delete - Lazy*

O **Scan-queue** faz buscas na *Control-queue*, executando retiradas de elementos da fila até que um objeto S que foi pintado de preto seja encontrado. Um *mark-scan* local é então executado sobre o sub-grafo S.

```
Scan-queue() =
    S = front-of-Control-queue
    Pop-Control-queue
    if (colour(S) == black) then
        broadcast Suspend
        Mark-red(S)
        Scan(S)
        Collect(S)
        broadcast Continue
    else
      if (Control-queue not empty) then
          Scan-queue()
      else
          Msg “No pages found”
```

Algoritmo 4.14 – Fase *Scan-queue - Lazy*

Novamente, por simplicidade, deve haver uma condição de que toda a computação seja suspensa durante a fase do *mark-scan*. O *Mark-red*(S) é modificado para tratar as novas páginas *black*:

```
Mark-red(S)=                                              (in Server S)
    if (colour(S) == green or black) then
        colour(S)= red
        for T in Sons(S) do
            send Message-mark-red <S,T> to T
```

Algoritmo 4.15 – Fase *Mark-red - Lazy*

Os algoritmos *Handle-mark-red, Scan* e *Collect* ainda têm comportamento de execução imediata.

4.4.1 Processamento durante a fase *mark-scan*

Até agora, consideramos que toda computação fosse suspensa durante a fase do *mark-scan* local. Embora haja perda de eficiência, esta não é tão alta quanto seria na execução do *mark-scan* global. A seguir descrevemos como fazer para que esta condição seja “relaxada” a fim de que o processamento continue durante a fase *mark-scan*. Descreveremos somente o

algoritmo *Mark-Scan Lazy* Ponderado, os outros são apenas descritos forma direta. O requisito essencial para qualquer relaxamento é que ele mantenha o relacionamento entre os contadores de referência de um objeto e os pesos de todos os ponteiros para este objeto. Isto é muito importante, uma vez que o *mark-scan* tenha sido iniciado, nenhum outro poderá ser executado até que o primeiro tenha terminado – ou seja, deve ser uma transação atômica.

O **New(P**) continua o mesmo, diferentemente do algoritmo de coleta de lixo de células de memória.

O algoritmo de cópia de um hiperlink tem a ação de dividir seu peso em duas partes iguais entre o original e a cópia. Não há comunicação com o alvo do hiperlink. Em particular, não é possível copiar um hiperlink (com seu peso dividido) de uma página que está envolvida em um *mark-scan* local até que ele seja completado a varredura em todas as páginas envolvidas. Sendo assim, o algoritmo de cópia é apenas corrigido como se segue:

```
Copy(R,<S,T>) =
     while (colour(R) or colour(S) == red) wait
           make hiperlink <R,T>
           W(<R,T>) = W(<S,T>) / 2
           W(<S,T>) = W(<R,T>)
```

Algoritmo 4.16 – Operador *Copy* durante a fase *mark-scan - Lazy*

O algoritmo para o *Delete* permanece o mesmo, mas a suspensão é agora relaxada para ser menos restritiva, suspendendo apenas a capacidade de outros servidores de iniciar o *mark-scan*. No entanto, o *Scan-queue* pode ser bloqueado enquanto um *mark-scan* estiver em execução. Igualmente, pode-se garantir que somente um processo por vez pode chamar o *Scan-queue*. Comete-se erro ao observar que estas duas condições significam a possibilidade de dois processos executarem *mark-scans* locais nos mesmos sub-grafos com conseqüências indesejadas. Se a *Control-queue* não está cheia, o *Delete* é então direcionado para proceder de forma normal, mesmo durante o *mark-scan*. Com essas modificações pode-se continuar com todas as partes do grafo durante o *mark-scan* local, a exceção quando:

- Uma tentativa de se fazer uma alteração na conectividade do sub-grafo estando o mark-scan em ação para copiar um ponteiro;

- Um hiperlink para compartilhar uma página é excluído quando a *Control-queue* está cheia;

4.4.2 Evitando esperas (*delays*)

Jones e Lins **[JL93]** propuseram um algoritmo de contagem de referência ponderada para tratar estruturas cíclicas em um ambiente de processamento paralelo. No entanto, Plainfossé e Shapiro **[PS95]** avaliaram este algoritmo e afirmaram que ele não é totalmente concorrente, uma vez que pode haver um cenário que seja desfavorável no qual parte da rede pode ser bloqueada e esperar que outro processador finalize sua operação. Esta deficiência foi corrigida por Jones e Lins no mesmo ano através de uma solução proposta similar adotada por Lins em arquitetura de memória compartilhada. Barreiras de sincronização são também adotadas depois de cada fase de *mark-scan*. Isto garante que coletas de lixo simultâneas não interfiram entre si e que por meio disso perdiam a cor e a informação do peso. Se um mecanismo é implementado tal como servidores inativos que pode capturar páginas para o *mark-scan* de um pool de páginas, a rede fica com um alto fluxo de saída. Assim, o *Scan-queue* verifica a *Control-queue*, retirando elementos da fila até que uma página S que tenha sido pintada de *black* seja encontrada. Um *mark-scan* local é então executado no sub-grafo S, sincronizando depois de cada fase, com se segue:

```
Scan-queue() =
     S = front-of-Control-queue
     Pop-Control-queue
     if (colour(S) == black) then
         RENDEZVOUS
         Mark-red(S)
         RENDEZVOUS
         Scan(S)
         RENDEZVOUS
         Collect(S)
     else
         if (Control-queue not empty) then
            Scan-queue()
         else
            Msg “No pages found”
```

Algoritmo 4.17 – Fase *Scan-queue* – evitando espera

Segundo Jones e Lins **[JL96]**, um segundo contador de referência (SRC) faz parte de cada célula e é usado somente pelo coletor de lixo. Isto garante maior independência entre processadores mesmo tendo pouco espaço.

4.4.3 O algoritmo *mark-scan* local – Considerações

Este algoritmo pode ser visto como uma mesclagem entre os algoritmos de contagem de referência ponderada e os algoritmos de contagem de referência cíclica com *mark-scan* local. As únicas diferenças visíveis entre eles são:

- *colour*(U) := *green* + a instrução *New*(P). A contagem de referência ponderada ignora a cor da informação;
- A cláusula **else,** que executa o *mark-scan* no *Handle-Delete*, é a responsável pela detecção de ciclos.

Como nenhum dos algoritmos acima altera a informação do peso, eles podem ser considerados com uma extensão conservativa do algoritmo de Contagem de Referência Ponderada.

A operação ***Suspend*** transforma modelo de processamento distribuído com páginas distribuídas em um modelo de único servidor com um URL simples e, além disso, a arquitetura apresentada pode ser vista como um único servidor com um URL local. Esta

arquitetura para o qual o algoritmo de contagem de referências cíclicas com *mark-scan* local já foi provado correta por Martinez-Wachenchauzer e Lins **[MWL90]**. A operação ***Continue***, que fica no final do *mark-scan*, permite que os processos resumam a computação de forma que seja restaurada a distributividade da arquitetura.

Este novo algoritmo somente difere do *mark-scan* original no uso de pesos. No entanto, o objetivo é o mesmo: executar o *mark-scan* local de forma que seja permitida a exclusão de uma referência em uma estrutura compartilhada.

A idéia de suspender o processamento local leva a falsa impressão de se tem que paralisar todas as operações da Web como um todo. Parar a rede mundial seria algo inaceitável. No entanto, o objetivo dos algoritmos que utilizam barreiras de sincronização é justamente limitar a parada ao contexto local dos servidores envolvidos com as páginas vinculadas.

4.4.3.1 O algoritmo *mark-scan* ponderado *lazy*

Este algoritmo é basicamente o mesmo que o *mark-scan* local original, a não ser pelo fato de que posterga sua execução o mais tarde possível enquanto que o outro é executado toda vez que um ponteiro é excluído. O gerenciamento da *Control-queue* é exatamente o mesmo no caso dos algoritmos para um único processador.

4.4.3.2 Processamento durante o *mark-scan*

Os algoritmos que suspendem o processamento durante a fase do *mark-scan* nos dão simplicidade e segurança, mas nos impõe atrasos e perda de desempenho no sistema. A idéia básica desses algoritmos é imitar o comportamento do *mark-scan* de forma que qualquer processo que modifique uma célula (uma página Web no contexto desta dissertação) que está na área do deste, deveria se comportar da mesma forma como se tivesse acontecido antes daquele. Vamos analisar aquelas instruções usadas durante o processamento *do mark-scan*:

New: herda uma célula nova de seu pai e a trata como se fosse parte do grafo antes do *mark-scan*. Copiando a cor da célula pai e zerando o contador de referências de uma célula

red enquanto cores de células pais sejam *red* ou *blue*, é equivalente ao se fazer um *mark-scan* no novo sub-grafo que está sendo formado apenas por células novas.

Copy: se auto-suspende apenas quando o *mark-scan* está sendo executado em uma das células pais.

Delete: é seguro enquanto a rede não providencia nenhuma nova chamada ao *mark-scan*. Blocos de *Scan-queue* se enquadram nessa mesma situação.

Capítulo 5

Conclusões e trabalhos futuros

As páginas Web por estarem em um ambiente distribuído e por fazerem referências entre si por meio dos hiperlinks, se mostram notoriamente adequadas para a aplicação de algoritmos de coleta de lixo com contador de referência ponderada distribuída, incluindo o tratamento de ciclos que ocorrerem com freqüência. Os algoritmos aplicados ao gerenciamento das páginas Web se mostram bastantes promissores.

Para arquitetura com múltiplos processadores, a contagem de referência ponderada se mostra simples e com eficiente gerenciamento de páginas. Através da associação de pesos com hiperlinks, comunicações entre processos nos servidores só entram em ação quando houver exclusão de algum hiperlink compartilhado. Hiperlinks podem ser copiados sem que haja necessidade de comunicação com o proprietário da página. Muitos algoritmos estenderam as técnicas de contagem de referência para permitir a reclamação de estruturas cíclicas em monoprocessadores e em arquiteturas paralelas fortemente acopladas. Os novos algoritmos de Lins apresentados aqui combinam eficiência e técnicas de contagem de referência ponderada e contagem de referência cíclica.

A contagem de referência cíclica requer que seja executado um *mark-scan* local a fim de reclamar estruturas cíclicas. O *mark-scan* é por herança uma ação atômica, mas o algoritmo para contagem de referência cíclica ponderada apresentada permite o processamento como um todo continuar durante a fase de busca, exceto nos casos:

- Quando há uma tentativa de se fazer uma cópia de um hiperlink que faz parte de um sub-grafo que está sofrendo uma operação de *mark-scan*;
- Quando um hiperlink para uma página distribuída é excluído quando a *Control-queue* está cheia.

Nestas situações, o processador faz uma requisição de espera até que o *mark-scan* finalize. Em todos os outros casos o processamento tem continuidade.

Este novo algoritmo se comparado ao de contagem de referência ponderada tem apenas um acréscimo de um pequeno overhead de espaço que está disponibilizado para armazenar a cor da página e um espaço para armazenar a *Jump-stack* e a *Control-queue*. A sobrecarga agregada ao processamento e à comunicação do *mark-scan* é proporcional ao número de hiperlinks compartilhados excluídos. Este custo pode ser bastante reduzido com a aplicação do algoritmo *lazy*, onde poucas são as chamadas diretas ao *mark-scan*. O custo-benefício de se reclamar as estruturas cíclicas pode justificar estas pequenas perdas.

Como a World Wide Web está em constante evolução, sempre estão surgindo novas propostas de melhoras apresentadas a W3C, a proposta deste trabalho pode ser apresentada como uma nova RFC para a linguagem HTML.

Pode-se definir um protocolo robô para que ele faça a verificação para evitar que determinada página seja excluída arbitrariamente sem que seu contador de referência esteja zerado. Esta definição pode ser uma adaptação dos protocolos existentes.

Uma nova definição para as linguagens dos servidores de páginas Web seria necessário para inclusão e processamento do controle interno do contador de referências ponderado.

Um experimento pode ser feito em uma rede local com alguns servidores de páginas Web que possuem URLs diferenciados. No entanto, para tal, como não há a implementação das mudanças nas linguagens HTML e *script* dos servidores de páginas, o que se poderia fazer era definir e implementar um mini-compilador *script*. Para que se realize uma simulação com a definição atual do HTML com esse mini-compilador, poderia-se utilizar uma diretiva de compilação em formato de comentário (<!-- isto é um comentário em HTML -- >) que contivesse os contadores em um arquivo de extensão “**.css**”.

Referências bibliográficas

[AM98] AROCENA, Gustavo; MENDELZON, Alberto. *WebOQL: Restructuring documents, databases and Webs*. In Int. Conf. On Data Engineering, p. 24-33, Orlando Florida, 1998.

[App87] APPEL, Andrew W. *Garbage Collection can be faster than stack allocation*. Information Processing Letters 25, pp.275-279, Jun 1987.

[BC92] BOEHM, H.; CHASE, D.R. *A Proposal for Garbage-Collection-Safe C Compilation*. Journal of C Language Translation, p.126-141, 1992.

[BEN+93] BIRRELL, A.; EVERS, D.; NELSON, G.; OWICKI, S.; WOBBER, E. *Distributed Garbage Collection for Network Objects*. TR116, Digital Systems Research Center, Palo Alto, CA, Dec 1993.

[Bev87] BEVAN, D. I. *Distributed garbage collection using reference counting*. In PARLE 1987, Parallel Architectures and Languages Europe, Volume II: Parallel Languages, published as: Lecture Notes in Computer Science 259, pages 176-187, Jun 1987.

[Bis77] BISHOP, P.B. *Computer systems with a very large address space and garbage collection*. MIT Report LCS/TR-178, Laboratory for Computer Science, MIT, May 1977.

[Bus45] BUSH, Vannevar. *As we may think*. Atlantic Monthly, 1945. Also in Computer-Supported Cooperative Work: A Book of Readings, ed. I. Greif, Morgan-Kaufmann, San Diego, CA 1998.

[Che70] CHENEY, C. J. *A non-recursive list compacting algorithm*. Communication of the ACM, 13(11): 677-678, Nov 1970.

[Col60] COLLINS, G. E. *A method for overlapping and erase of lists*, Comm. of ACM, Vol. 3(12):655-657, Dec 1960.

[Com01] COMER, Douglas E. *Computer Networks and Internets, with Internet Applications*. 2sd. Edition. Prentice-Hall, 2001.

[Con87] CONKLIN, J. *A Survey of Hypertext*. Technical Report STP-356-86, Microelectronics and Computer Technology Corporation, Austin, Texas, Feb 1987.

[CV+90] CORPORAAL, H.; VELDMAN, T.; VAN DE GOOR, A. J. *An Efficient Reference Weight-Based Garbage Collection Meted for Distributed Systems.* In Proceedings of the PARBASE-90 Conference, pages 463-465. IEEE, 1990.

[DB76] DEUTCH, L. P.; BOBROW, D.G. *An efficient incremental automatic garbage collector*. Communications of the ACM, 19(7), July 1976.

[Dic92] DICKMAN, Peter. *Optimising Weighted Reference Counts for Scalable Fault-Tolerant Distributed Object-Support Systems.* Submitted for HICSS 26, June 1992.

[DL+78] DIJKSTRA, E. W.; LAMPORT, L.; MARTÍN, A. J.; SCHOLTEN, C.S.; STEFFENS, E.F.M. *On-the-fly Garbage Collection: An Exercise in Cooperation*. Communications of the ACM, 21(11): p. 965-975, Nov.1978.

[Fer+97] FERNANDEZ, Mary; FLORESCU, Daniela; LEVY, Alon; SUCIU, Dan. *A Query Language for a Web-site Management System*. SIGMOD Record, 26(3):4-11, Sep 1997.

[FY69] FENICHEL, Robert R.; YOCHELSON, Jerome C. *A LISP Garbage Collection for virtual memory computer systems*. Communications of the ACM, 12(11):611-612, Nov. 1969.

[GB+01] GRUNE, Dick.; BAL, Henri E.; JACOBS, Ceriel J.H.; LANGENDOEN, Koen G.. *Modern Compiler Design*. 1st Edition. John Wiley & Sons, 2001.

[Glu89] GLUSHKO, R. J. *Design Issues for Multi-Document Hypertexts*. In Hypertext'89 Proceedings. Pittsbrugh, Nov-1989.

[Gol90] GOLDFARB, Charles. *The SGML Handbook*. Oxford University Press, Oxford, 1990.

[Hin+97] HIMMERODER, Rainer; LAUSEN, Georg; LUDASCHER, Bertram; SCHLEPPHORST, Christin. *On a Declarative Semantics for Web Queries*. In Proc. of the Int. Conf. on Deductive and Object-Oriented Database (DOOD), p. 386-398, Singapore, Dec 1997.

[HK96] HARRIS, Stuart; KIDDER, Gayle. *Official Html Publishing for Netscape: Windows Edition*. Ventana Communications Group Inc.; Book and CD edition - June 1996.

[Hug85] HUGHES, John. *A distributed garbage collection algorithm*. In Jean-Pierre Jouannaud, editor, ACM Conference on Functional Programming Lan guages and Computer Architecture, number 201 in Lecture Notes in Computer Science, pages 256--272, Nancy, France, Software Practice and Experience 1985. Springer-Verlag.

[JL93] JONES, Richard; LINS, Rafael. *Cyclic Weighted Reference Counting*. In K. Boyanov (ed.), Proc. of Intern. Workshop on Parallel and Distributed Processing, NH, May 1993.

[JL96] JONES, Richard; LINS, Rafael. *Garbage Collection – Algorithms for Automatic Dynamic Memory Management*. 1st Edition. New York: John Wiley & Sons, 1996. (Revised edition in 1999).

[KR01] KUROSE, J.A.; ROSS, K. W. *Computer Networking: a top-down approach featuring the Internet*. Reading, MA: Addison-Wesley, 2001.

[Lin02a] LINS, Rafael Dueire. *An Efficient Algorithm for Cyclic Reference Counting*. Information Processing Letters, vol 83(3):145-150, Aug 2002.

[Lin02b] LINS, Rafael Dueire. *Efficient Cyclic Weighted Reference Counting*. 14th SBAC–PAD: p. 61-67, October 28 - 30, 2002 – Vitória-ES, Brazil; IEEE Press.

[Lin92] LINS, Rafael Dueire. *Cyclic reference counting with lazy mark-scan*. Information Processing Letters, 44(4): 215-220, North-Holland, Dec 1992.

[LQP92] LANG, B.; QUENNIAC, C.; PIQUER, J. *Garbage Collection the World*. ACM Symposium on Principles of Programming, Albuquerque, 1992.

[Mar63] MINSKY, Marvin. *A LISP garbage collector algorithm using serial secondary storage*. Technical Report Memo 58 (rev.), Project MAC, MIT, Cambridge, MA, Dec 1963.

[McB63] McBETH, J. H. *On the Reference Counter Method*. Communications of the ACM, 6(9):575, Sep 1963.

[McC60] McCARTHY, John. *Recursive functions of symbolic expressions an their computation by machine*. CACM, 3:184-195, 1960.

[MWL90] MARTINEZ, A. D.; WACHENCHAUZER, R.; LINS, R. D. *Cyclic Reference Counting with Local Mark-Scan*. Inform. Process. Lett, 34: p. 31-35, North-Holland, 1990.

[Piq96] PIQUER, J. M. *Indirect Distributed Garbage Collection: Handling Object Migration*. ACM Transactions on Programming Languages and Systems, Vol.13, No.3, Sep 96.

[PS95] PLAINFOSSE, D.; SHAPIRO, M. *A Survey of Distributed Garbage Collection Techniques*. International Workshop on Memory Management, Kinross, UK, Sep 1995.

[PU03] PASQUALE & ULISSES. *Gramática da Língua Portuguesa*. Nova Edição Reformulada. São Paulo, Scipione, 2003.

[Rad91] RADA, R. *Hypertext: FromText to Expertext*. New York; McGrall-Hill, 1991.

[RJ96] RODRIGUES, H.; JONES, R. *A Cyclic Distributed Garbage Collector of Network Objects*. International Workshop on Distributed Algorithms (WDAG), 1996.

[Rov85] ROVNER, P. *On Adding Garbage Collection and Runtime types to a strongly-typed, statically checked, concurrent language*. Technical Report CLS84-7, Xerox PARC, Palo Alto, CA, Jul 1985.

[Sal92] SALGADO, Ana Carolina et al. *Sistemas Hipermídia: Hipertexto e Bancos de Dados*. VII Escola de Computação. Porto Alegre: Instituto de Informática da UFRGS, 1992.

[Sal02] SALZANO FILHO, José Airton. A*lgoritmos para Contagem de Referências Cíclicas*. Dissertação de Mestrado. Universidade Federal de Pernambuco, 2002.

[SK89] SHNEIDERMAN, Ben; KEARSLEY, Greg. *Hypertxt Hands-On!: An Introduction to a New Way of Organizing and Accessing Information*. Reading, MA: Addison-Wesley, 1989.

[SKS02] SILBERSCHATZ, Abraham; KORTH, Henry F.; SUDARSHAN, S. *Database System Concepts*. 4th Edition. New York, McGraw-Hill, 2002.

[Sou00] SOUZA, Francisco Vieira de. *Aspectos de Eficiência em Algoritmos para Gerenciamento Automático de Memória*. Tese de Doutorado. Universidade Federal de Pernambuco, 2000.

[Str91] STROUSTRUP, Bjarne. *The C++ Programming Language.* Addison Wesley, 2sd Edition, 1991.

[Tan03] TANENBAUM, Andrew S. *Computer Networks.* 4th Edition. New Jersey, Prentice Hall, 2003.

[Wen79] WENG**,** K. S. *An Abstract Implementation for a Generalized Dataflow Language*. MIT Laboratory for Computer Science MIT/LCS/TR-228. 1979.

[Wil94] WILSON, P.R. *Uniprocessor Garbage Collection Techniques*. Technical Report, University of Texas, Jan 1994.

[WW87] WATSON, Paul; WATSON, Ian. *An Efficient Gargabe Collection Scheme for Parallel Computer Architectures.* In PARLE 1987, Parallel Architectures and Languages Europe, Volume II: Parallel Languages, published as: Lecture Notes in Computer Science 259, pages 432-443, June 1987.

[YN99] YATES, R. B.; NETO, B. R. *Modern Information Retrieval*. ACM press; New York, Addison-Wesley, 1999.

[ZD98] ZIFF-DAVIS Publishing. "*Ted Nelson: Hypertext pioneer*". http://www.zdnet.com/zdtv/screensavers_story/0,3656,2127396-2102293,00.html. 1998.

[Zor93] ZORN, Benjamin. *The Measured Cost of Conservative Garbage Collection*. Software-Practice and Experience, 23(7):733-756, 1993.

[1st] 1st Page 2000 – EVR Soft: http://www.evrsoft.com/; Dez 2003.

[ALG58] Algol-58 (Algorithmic Language) – International committee, 1958.

[Ace] AceHTML – Visicom Media: http://freeware.acehtml.com/; Dez 2003.

[AV] Altavista home page: http://www.altavista.com; Dez 2003.

[C72] C - Kenneth Thompson and Dennis Ritchie - AT&T Bell Laboratories, 1972.

[COB60] COBOL (Common Business-Oriented Language) – CODASYL committee, 1960.

[CP7] Cool Page: http://www.coolpage.com/cpg.html; Dez 2003.

[Cad] Cadê home page: http://www.cade.com.br; Dez 2003.

[DEM] DICIONÁRIO ELETRÔNICO MICHAELIS – UOL, em CD instalável, 2002.

[Dom] DominHTML - Domino: http://www.dominocs.com/DHTML/; Dez 2003.

[Dre] Macromedia Dreamweaver: http://www.macromedia.com/software/dreamweaver/; Dez 2003.

[FOR57] FORTRAN (Formula Translating) – John Backus - IBM, 1957.

[Goo] Google home page: http://www.google.com; Dez 2003.

[Has] Haskell – A Purely Functional Language: http://www.haskell.org/; Jan 2004.

[HD] HotDog - Sausage Tools: http://www.sausagetools.com/; Dez 2003.

[HS] HomeSite: http://www.macromedia.com/software/homesite/; Dez 2003.

[Jav95] Java – James Gosling et al. - Sun Microsystens,1995. Java home page: http://java.sun.com; Dez 2003.

[LIS58] LISP (List Processing) - John McCarthy – MIT, 1958.

[Mos] NCSA Mosaic: http://archive.ncsa.uiuc.edu/SDG/Software/Mosaic/NCSAMosaicHome.html ; Dez 2003.

[MSIE] Microsoft Internet Explorer: http://www.microsoft.com/windows/ie/default.asp; Dez 2003.
Download: http://www.microsoft.com/windows/ie/downloads/critical/ie6sp1/default.asp ; Dez 2003.

[MSFP] Microsoft Office FrontPage: www.microsoft.com/frontpage; Dez 2003.

[Net] Netscape: http://www.netscape.com/
Download: http://channels.netscape.com/ns/browsers/download; Dez 2003.

[Sot] Sothink HTML Editor: http://www.sothink.com/; Dez 2003.

[TAM45] The Atlantic Monthly (Magazine). *As we may think*. July, 1945. Volume 176, n. 1; 101-108.
Also in http://www.theatlantic.com/unbound/flashbks/computer/bushf.htm; Dez 2003.

[Tex] *LaTex* – A Document Preparation System: http://www.latex-project.org/; Dez 2003.

[VP] Vanish Project: http://www.cgl.uwaterloo.ca/Projects/Vanish/webquery-1.html; Dez 2003.

[W3C-a] World Wide Web Consortium: http://www.w3c.org/TR/1998/REC-html40-19980424/intro/intro.html - A brief history of HTML; Dez 2003.

[W3C-b] World Wide Web Consortium: http://www.w3c.org/Style/CSS/ - Cascading Style Sheets (CSS); Dez 2003.

[W3C-c] World Wide Web Consortium: http://www.w3c.org./History.html - A Little History of the World Wide Web; Dez 2003.

[W3C-d] World Wide Web Consortium: http://www.w3c.org./MarkUp.html – HyperText MarkUp Luanguge – HTML – Home page; Dez 2003.

[WML] The Web Modeling Language: http://www.webml.org/webml/page1.do - WebML Org; Dez 2003.

[WR] Web Ratio home page: http://www.webratio.com/; Dez 2003.

[Yah] Yahoo home page: http://www.yahoo.com; Dez 2003.

Glossário

Acíclico	Que não forma ciclo; incapacidade de gerenciar estrutura cíclica;
Alocação	Aquisição de espaço realizado pelo gerenciador de memória;
Alocação dinâmica	Alocação padrão onde a localização e o espaço ocupado na memória de todos os dados são determinados em tempo de execução;
Alocação estática	Alocação padrão no qual se conhece a localização e o layout de todos os dados determinados em tempo de compilação;
Alocação na heap	Alocação de objetos em uma área de memória não sujeita à disciplina LIFO da alocação de pilha (stack);
Alocação de pilha (stack)	Alocação padrão que segue a disciplina LIFO (o último a entrar é o primeiro a sair);
Arquitetura Paralela Fortemente Acoplada	Arquitetura onde todos os processadores compartilham o mesmo espaço de memória;
Arquitetura Paralela Fracamente Acoplada	Arquitetura onde cada processador possui o seu próprio espaço de memória;
Assíncrono	Que não é executado ao mesmo tempo (simultaneamente);
Barreira de escrita	Barreira que impede a escrita de um objeto;
Barreira de leitura	Barreira que impede a leitura de um objeto;

Bitmap Array de bits. Tipicamente usado por coletores de lixo para marcação, onde cada bit representa uma palavra ou um objeto na heap;

Broadcast Processo de transmissão de informação para todos os pontos alcançáveis;

Célula Uma unidade dos campos contíguos de memória que formam uma unidade lógica;

Célula atômica Célula que não contém um ponteiro;

Ciclo Um subconjunto de estrutura de dado ligada que forma uma corrente fechada, no qual qualquer célula pode ser localizada a partir de uma outra célula qualquer do subconjunto;

Coletor de lixo Processo ou processador responsável pela coleta de lixo; algoritmo que recicla automaticamente o lixo;

Compreensivo Propriedade de um algoritmo de coleta de lixo segundo a qual todo lixo só é reclamado ao fim de seu ciclo de coleta;

Concorrente Diz-se que dois processos são concorrentes se eles podem ser executados de modo assíncrono sem qualquer intercalação pré-definida;

Conservativo Um algoritmo de coleta de lixo que superestima a porção de dados vivos. Diz-se especialmente o coletor de lixo que espera uma pequena cooperação do compilador (e em particular que não tenha nenhum conhecimento sobre a localização dos objetos) e o coletor incremental e concorrente que adia a reclamação de algum lixo até o próximo ciclo;

Conveniência (expediency) Propriedade de um algoritmo de coleta de lixo que pode reclamar lixo ainda que parte do sistema de distribuído esteja indisponível;

Dado ativo Dado em uso; oposto ao dado livre ou lixo;

Desalocação Retorno do espaço ocupado ao gerenciador de memória; desocupação do espaço de memória;

Desalocação explícita Desalocação realizada via comando de programação;

Espaço de endereçamento Intervalo de valores que um ponteiro mantém;

Filho (child) Uma célula B é dita ser um filho da célula A se A mantém um ponteiro para B;

Finalização Ação de limpeza e liberação de recursos executada sobre um objeto quando ele morre;

Fragmentação Propriedade que caracteriza a situação em que se encontra a heap quando ela não está completamente ocupada e que ainda contém espaço suficiente total para nova alocação, mas esta não é permitida; propriedade que caracteriza uma memória "esburacada" (cheia de espaços em branco entre espaços ocupados);

Free-list Uma lista ligada de células livres;

Heap Região da memória no qual os objetos são alocados em uma ordem não especificada;

Host Computador que hospeda uma aplicação ou objeto;

Incremental	Diz-se o algoritmo no qual a computação é executada em passos incrementais (aos poucos) entre uma parada e outra;
Livre (free)	Célula que está disponível para reuso; retorno de uma célula não utilizada ao gerenciador de armazenamento;
Lixo	Espaço de memória não mais utilizado por nenhum processo, mas que ainda não foi reclamado pelo gerenciador de memória ou coletor;
Mutador	Processo ou processador responsável pela execução dos processos de usuários que particularmente altera a conectividade do grafo de objetos;
Ponteiro	Variável que contém um endereço ou referência de uma célula de memória;
Objeto	Uma célula; instância de uma classe;
Operação atômica	Operação que uma vez iniciada será completada sem interrupção;
Processo ativo	Processo em execução;
Raiz	Localidade de memória a partir do qual sempre é possível localizar um objeto;
Reclamação	Ação de retorno do lixo ao gerenciador de memória para futura reutilização do mesmo;
Registro de ativação	Registro que guarda o estado da computação atual e retorna seu

	endereço de memória;
Rendezvous	Ponto de sincronização; quando dois processos (ou mensagens) aguardam um instante para sincronização;
Síncrono	Que é executado simultaneamente;
Vivo	Dado ou objeto que ainda está ligado direta ou transitivamente à raiz; dado ou objeto ativo;