UNIVERSIDADE FEDERAL DA PARAIBA

CENTRO DE CIENCIAS E TECNOLOGIA

DEPARTAMENTO DE SISTEMAS E COMPUTAÇÃO

COORDENAÇÃO DE PÓS-GRADUAÇÃO EM INFORMATICA

CURSO DE MESTRADO EM INFORMATICA

COMPRESSÃO/DESCOMPRESSÃO DE DADOS

Campina Grande
Abril - 1989

COMPRESSAO/DESCOMPRESSAO DE DADOS

- UMA VARIAÇAO DO ALGORITMO LZW -

Francisco Vilar Brasileiro

COMPRESSAO/DESCOMPRESSAO DE DADOS

- UMA VARIAÇAO DO ALGORITMO LZW -

Dissertaçao submetida ao Curso de MESTRADO EM INFORMATICA da Universidade Federal da Paraíba, em cumprimento ás exigências para a obtenção do Grau de Mestre.

AREA DE CONCENTRAÇAO: **Software Básico**

José Antão Beltrão Moura
Orientador

Jacques Philippe Sauvé
Co-orientador

Campina Grande
Abril - 1989

B823c Brasileiro, Francisco Vilar
Compressao/descompressao de dados : uma variacao do algoritmo LZW / Francisco Vilar Brasileiro. - Campina Grande, 1989.
106 f. : il.

Dissertacao (Mestrado em Informatica) - Universidade Federal da Paraiba, Centro de Ciencias e Tecnologia.

1. Algoritmo de Compressao 2. Algoritmo LZW 3. Dissertacao I. Moura, Jose Antao Beltrao II. Sauve, Jacques Philippe III. Universidade Federal da Paraiba - Campina Grande (PB) IV. Título

CDU 004.421(043)

COMPRESSAO/DESCOMPRESSAO DE DADOS

- UMA VARIAÇAO DO ALGORITMO LZW -

Francisco Vilar Brasileiro

Dissertação aprovada em: 24 de abril de 1989

José Antão Beltrão Moura
Orientador

Jacques Philippe Sauvé
Co-orientador

Marcos Antonio Gonçalves Brasileiro
Examinador

Maria de Fátima Camelo
Examinadora

Campina Grande
Abril - 1989

Ao Deus que dá sentido a existência

Aos meus pais, que me fizeram ver o valor da educação

AGRADECIMENTOS

Aos professores orientadores Antão Moura e Jacques Sauvé, que foram "verdadeiras tochas", ora na frente iluminando o caminho, ora na retaguarda não permitindo recuos.

A Infocon Software, sem a qual, a realização desta tarefa teria sido muito árdua.

A Coordenação de Pós-Graduação em Informática - COPIN, pelo incentivo e apoio em todos os momentos.

Aos colegas do mestrado, pela amizade e companheirismo.

A Valéria, que esteve muito presente.

A Lika, pelo carinho comigo, e a ajuda na "reta final".

SUMARIO

Descreve-se um algoritmo de compressão em um passo que explora localidade de referência, como a que ocorre quando padrões são usados frequentemente em uma determinada porção de mensagens ou arquivos e depois caem em desuso. O algoritmo é uma modificação do algoritmo LZW apresentado por Welch. Medidas obtidas com arquivos reais, apontam um melhor desempenho frente às técnicas LZW e de Huffman.

ABSTRACT

A one-pass compression algorithm that exploits locality of reference, such as occurs when data patterns are used frequently over certain portions of messages or files and then fall in disuse, is described. The algorithm is a modification of the algorithm LZW by Welch. Measurements with real files indicate that it performs better than LZW and Huffman coding.

Indice

Indice de figuras

CAPITULO I

Introdução

O computador tem se mostrado uma ferramenta bastante utilizada nas mais diversas áreas do conhecimento humano. A disseminação do uso de computadores na automação das atividades científicas, comércio-industriais, econômicas e militares impõe uma natural expansão da quantidade de informação que deve ser armazenada e acessada para suporte destas aplicações.

Os grandes bancos de dados, as redes de computadores e o teleprocessamento em geral permitiram o surgimento de aplicações distribuidas. A proliferação destas aplicações contudo, demanda altas capacidades de armazenamento de informação e canais de comunicação com taxas de transferência cada vez maiores.

A demanda cresce com a expansão da quantidade de informação a ser processada. Faz-se pois necessário, buscar tecnologias que resolvam ou aliviem o problema desta demanda.

1.1 Avanços nas tecnologias de armazenamento e de comunicações

As unidades de armazenamento a laser e os canais de comunicação com fibras óticas são evoluções na tecnologia de hardware para resolver o problema acima discutido. Entretanto, a explosiva proliferação da informação e o crescimento continuo das necessidades das aplicações, poderão ultrapassar qualquer desenvolvimento tecnológico alcançado na área de armazenamento e transmissão de dados.

Uma alternativa à aquisição de hardware para armazenamento e comunicações é a adoção de técnicas que reduzam o tamanho dos dados a serem efetivamente armazenados e/ou transmitidos. É possivel que exista um certo grau de redundância nos dados manipulados, permitindo que a informação contida nos dados possa ser representada de uma maneira mais econômica, através da diminuição dessa redundância. A diminuição de redundância na representação da informação pode ser alcançada pelas técnicas de compactação e de compressão de dados.

1.2 Compactação e compressão de dados

Na compactação, elimina-se fisicamente informação irrelevante, preservando-se a integridade da representação da informação relevante. Ilustrando: os zeros à extrema esquerda em preços de produtos (por exemplo, NCz$ 000 1000,00) podem ser descartados sem prejuizo para o cálculo de faturamento (no caso, NCz$ 1000,00). O arquivo com dados de entrada para um aplicativo de controle de faturamento poderia pois, ser compactado (eliminado-se os zeros irrelevantes, no presente exemplo), reduzindo a área em disco por ele ocupada, ou baixando a utilização de linhas de comunicação quando de sua transferência.

Deve-se perceber que um conhecimento da semântica dos dados deve ser assumido, para que o compactador possa saber qual informação é ou não relevante. As técnicas de compactação são também conhecidas como técnicas não reversiveis, já que a informação original uma vez compactada, não poderá mais ser restaurada.

Na compressão, toda informação é considerada relevante e a redução física no tamanho dos dados é feita através de um mapeamento de códigos; o alfabeto fonte é mapeado em um alfabeto de códigos de forma a reduzir o número de bits da mensagem (informação) a ser comprimida. Um exemplo seria o mapeamento do preço de NCz$ 1000,00 em um código menor, por exemplo M. O processo inverso, de descompressão, recupera a representação original dos dados utilizando o mesmo mapeamento usado na compressão, ou seja, ao encontrar o código M o descompressor o mapeia para a sequência correspondente (NCz$ 1000,00). As técnicas de compressão são portanto reversíveis.

Métodos de compressão e compactação vem sendo estudados a muitos anos. A primeira solução para o problema de geração de códigos com redundância mínima (este conceito será discutido no capítulo II) foi proposto em um artigo clássico na literatura de compressão de dados, escrito por Huffman em 1952 [HUFF 52]. Huffman é considerado o pioneiro nesta área, não obstante alguns anos antes (1949), um algoritmo desenvolvido independentemente do algoritmo de Huffman, por Shannon e Fano [SHAN 49] [FANO 49], apresentasse uma solução para a geração de códigos com redundância mínima. O modelo apresentado por Shannon e Fano não garantia que os códigos gerados seriam sempre ótimos, entretanto, na maioria das vezes, uma razoável aproximação era conseguida.

A principal característica dos códigos de Huffman e Shannon-Fano é a sua simplicidade. Estes códigos têm sido alvo de bastante estudo e uma vasta literatura contendo variações e

aperfeiçoamentos destes códigos se encontra disponível [GALL 78] [MCIN 85] [TANA 87] [FALL 73] [VITT 87].

Os códigos apresentados por Shannon evoluiram para uma familia de códigos chamada Codificação Aritimética (arithmetic coding). Nestes códigos, ao invés de usar uma tabela de mapeamento para compressão e descompressão, o processo de compressão realiza operações aritméticas sobre a mensagem a ser comprimida, enquanto que o processo de descompressão realiza as mesmas operações de forma inversa.

Nos métodos da familia dos códigos de Huffman, a redundância dos dados é considerada de uma forma global. Um outro tipo de redundância, a redundância localizada em porções menores dos dados, não é tratada. Essa caracteristica dos dados deu origem a outra importante familia, a das técnicas OPM/L (Original Pointer Macro restricted to Left pointer). O capitulo II dará uma abordagem mais aprofundada destas e de outras técnicas de compressão e compactação.

1.3 Contribuição do trabalho

Bassiouni em [BASS 85] apresenta alguns pontos na área de compressão de dados que ainda não tiveram um tratamento adequado na literatura atualmente disponivel e sem dúvida necessitam de um estudo mais aprofundado.

O primeiro destes pontos é a inexistência de uma metodologia eficiente, que permita o desenvolvimento de modelos para comparação dos diversos métodos de compressão. Os modelos para compara-

ção de técnicas de compressão não devem ater-se apenas a confrontação de percentuais de compressão. Metodologias práticas devem considerar outros fatores como velocidade de compressão e descompressão, facilidade de implementação, quantidade de "overhead" com tabelas de mapeamento, complexidade na coleta de estatísticas para compressão, se necessário, entre outros.

Outra preocupação, oriunda do desenvolvimento crescente de sistemas distribuidos, reside no desenvolvimento de algoritmos de compressão para o propósito de transferência de informação. Nestes algoritmos, o processamento perdido na compressão pode ser aliviado pelo atraso inerente à transmissão da informação, permitindo a utilização de métodos mais sofisticados que obtenham um alto grau de compressão. Outro fator que deve ser estudado com cuidado neste tipo de algoritmo é a influência da confiabilidade do sistema de comunicação utilizado, na escolha de um algoritmo apropriado.

Outros pontos sugeridos é o desenvolvimento de implementações de métodos de compressão em hardware e desenvolvimento de algoritmos de compressão que atinjam maior grau de compressão e menor tempo de execução que os métodos atualmente disponiveis.

Este trabalho apresenta um algoritmo de compressão [BRAS 89a] [BRAS 89b], que por suas caracteristicas, velocidade e percentual de compressão atingidos, pode ser utilizado com êxito em transmissão de dados e na diminuição da quantidade de informação armazenada. As principais caracteristicas do algoritmo proposto estão listadas a seguir:

a. adequado para comunicação "on-line"

b. independente de contexto

c. percentual de compressão pouco variante com o tipo de informação a ser comprimida

d. adequado para compressão de grandes quantidades de informação

e. regenera a informação original

O algoritmo apresentado pertence a família dos códigos OPM/L e é uma extensão do algoritmo apresentado por Welch em [WELC 84]. Medidas obtidas com arquivos reais apontam um melhor desempenho do algoritmo proposto frente ao algoritmo de Welch.

A simplicidade de implementação e o tempo de processamento não são todavia afetados, permitindo que implementações em hardware, possam englobar as modificações aqui sugeridas à implementação apresentada por Welch.

A implementação em software, entretanto, não permite a utilização do algoritmo em aplicações em tempo real com alta velocidade (por exemplo, no "driver" de um disco).

O algoritmo, denominado LZW-MODIFICADO, está sendo utilizado para transferência de arquivos em uma ferramenta de comunicação entre sistemas UNIX e MS-DOS*.

* - UNIX é marca registrada da AT&T
MS-DOS é marca registrada da MicroSoft

1.4 Organização da tese

O restante desta tese é organizado em 5 capitulos.

O capitulo II faz uma revisão bibliográfica dos principais métodos de compactação e compressão de dados. Devido a sua pouca utilização e suas fortes limitações, a técnica de compactação não será tratada em detalhes.

Detalhes da especificação e implementação do algoritmo LZW-MODIFICADO são apresentados nos capitulos III e IV.

No capitulo V é feita uma avaliação do desempenho do algoritmo LZW-MODIFICADO frente a uma implementação do algoritmo LZW e uma implementação de um algoritmo baseado no código de HUFFMAN. Avaliações de possiveis variações na implementação do algoritmo LZW-MODIFICADO também são discutidas neste capitulo.

O último capitulo destina-se às conclusões e indicações de assuntos que necessitam de uma maior análise e discussão, e que portanto, constituem material para estudos futuros.

CAPITULO II

Técnicas de redução da redundância dos dados

Os métodos de redução da redundância dos dados, operam sobre a informação, tentando representá-la da forma mais econômica possivel. Antes de tratar dos métodos de redução da redundância dos dados propriamente, tentaremos formalizar alguns conceitos da teoria da informação, que serão importantes para uma melhor compressão do restante deste trabalho.

Seja E um evento que ocorre com probabilidade P(E). Se é sabido que o evento E ocorreu, então podemos dizer que a informação contida em E é expressa por:

$$I(E) = \log_r \frac{1}{P(E)} \text{ unidades de informação.}$$

Quando a base do logaritmo da expressão acima é 2 (r=2), temos a informação expressa em bits (binary unit). O conjunto binário de simbolos { 0, 1 } será assumido para representar os códigos por todo o trabalho, e por isso, "log" será sempre o logaritmo da base 2. Quando outro conjunto for usado, ele será explicitado.

Um código fonte é formado por um conjunto de simbolos S = { s_1, s_2, ..., s_n } com probabilidades $P(s_1)$, $P(s_2)$, ..., $P(s_n)$. Aqui, assumem-se os códigos considerados, como sendo sem memória, ou seja, a probabilidade de ocorrência de cada simbolo é independente da probabilidade de ocorrência dos demais.

Cada simbolo s_i do código fonte S contém $I(s_i) = \log 1/P(s_i)$ bits de informação. Como cada simbolo s_i tem probabilidade $P(s_i)$ de ocorrer, podemos definir a quantidade de informação média do código como sendo $\sum P(s_i)\ I(s_i)$ bits. A quantidade de informação média do código é a sua entropia H(S).

Os métodos de compressão discutidos na seção 2.2, se baseiam no mapeamento de códigos fontes em códigos comprimidos. Os códigos utilizados devem ser unicamente decodificáveis, ou seja, um mesmo código comprimido deve ser mapeado para uma única sequência de simbolos do código fonte, sob pena de não se poder recuperar a informação original. Outra caracteristica desejável para os códigos utilizados é que sejam instantâneos. Códigos instantâneos são decodificados sem a necessidade de informação extra ("lookahead").

Em cada caso, existem vários códigos que satisfazem as exigências apresentadas. A medida que é usada para escolher o código apropriado é, na maioria dos casos, o comprimento médio do código.

Seja $S = \{ s_1, s_2, \ldots, s_n \}$ um código fonte, que é mapeado no código $X = \{ x_1, x_2, \ldots, x_n \}$. Os elementos de X aprsentam probabilidades de ocorrência $P_1, P_2, \ldots, P_n$ respectivamente. Sejam $l_1, l_2, \ldots, l_n$ os comprimentos dos códigos de X (em bits). Definimos o comprimento médio (L) do código X como sendo:

$$L = \sum P_i\ l_i$$

O comprimento médio de um código é, na melhor das hipóteses, igual à entropia do código (H(S) $\leq$ L). Podemos definir a eficiência do código como sendo:

$$\text{eficiência} = \frac{H(S)}{L} .$$

A redundância do código por sua vez é definida como sendo:

redundância = 1 - eficiência [ABRA 63].

A entropia de uma mensagem indica o número mínimo de bits com os quais a mensagem pode ser codificada. Desta forma, um código é considerado ótimo se consegue codificar mensagens em um número de bits igual à entropia da mensagem. A entropia é portanto, uma grandeza bastante importante na avaliação do percentual de compressão de um método. Shannon foi quem primeiro apresentou estas expressões em [SHAN 49]. Uma abordagem mais simples e intuitiva pode ser encontrada em [ASH 65].

Os dados manipulados pelas aplicações por questões de clareza, simplicidade e desempenho, nem sempre se encontram representados através de um código que se aproxima da entropia dos dados, ou seja, através de um código de redundância mínima. Diversos tipos de redundância podem estar presentes nos dados a serem manipulados. Os principais tipos de redundância encontrados são: distribuição não homogênea dos caracteres, blocos com repetição de um mesmo caractere, repetição de sequências de caracteres, correlação, entre outros. A presença de um ou de outro tipo de redundância em uma massa de dados depende basicamente do tipo de informação que a mesma contém.

Dependendo do conteúdo do arquivo, os símbolos do código usado para representar os dados não se fazem presentes com a mesma frequência. Em um arquivo contendo texto é de se esperar que apenas os símbolos que representam as letras do alfabeto estejam presentes (além de alguns poucos símbolos especiais como ponto, virgula, etc.). Supondo que os dados sejam representados por um código de 8 bits (gerando um total de 256 símbolos), teríamos uma utilização de apenas 25% dos símbolos do código. Para representar os 64 símbolos utilizados, um código com apenas 6 bits seria suficiente, o que resultaria em uma diminuição de 25% no tamanho do arquivo.

Blocos com repetição de um mesmo caractere aparecem em diversas situações, podendo geralmente ser representados de uma forma mais compacta. Em arquivos contendo ilustrações é comum a existência de longas sequências de espaços em branco; arquivos contendo imagens apresentam regiões com grande número de "pixels" iguais; arquivos utilizados em processamento comercial com registros de tamanho fixo, geralmente apresentam sequências de caracteres de espaço em campos alfanuméricos não preenchidos totalmente e sequências de zeros em campos numéricos. Arquivos contendo texto por outro lado, raramente apresentam longas sequências de caracteres repetidos.

Alguns padrões de símbolos são encontrados mais frequentemente que outros, em determinados arquivos. As palavras chaves de uma linguagem de programação são exemplos deste tipo de redundância em arquivos contendo programas fonte. Em arquivos contendo textos, este tipo de redundância também é bastante comum.

Arquivos manipulados por aplicações comerciais geralmente apresentam campos que contêm um número reduzido de valores, de forma que valores iguais ao longo do arquivo ocorrem com bastante frequência.

A probabilidade de ocorrência de uma vogal em um arquivo de texto terá com certeza seu valor aumentado se for sabido de antemão que o último caractere lido foi uma consoante. A correlação entre os simbolos de um arquivo é um tipo de redundância dificil de medir, mas que pode ser utilizada para obter uma considerável redução no tamanho dos dados.

Uma macro classificação das técnicas de redução da redundância dos dados divide estas técnicas em dois grupos. Em um grupo se encontram as técnicas não reversiveis conhecidas como técnicas de compactação. Neste grupo não existe um processo que regenere os dados originais através dos dados compactados, embora estes representem a mesma informação contida nos dados originais. O outro grupo, o das técnicas reversiveis também denominadas técnicas de compressão, é formado pelas técnicas que se caracterizam pela necessidade de um processo que regenere a informação original. O processo necessário à decodificação da informação comprimida é a descompressão. Nas próximas seções serão discutidos os principais métodos de compressão e compactação.

2.1 Técnicas de compactação

Os métodos de compactação são bastante simples e baseiam-se na remoção de informação "inútil" dos dados. E facil notar que estes métodos são por definição dependentes do contexto ao qual os dados estão associados. Um grande conhecimento das caracteristicas não só dos dados, mas também das aplicações que manipulam os dados deve ser assumido.

A remoção de zeros à esquerda em campos numéricos e caracteres de espaço no final de campos alfanuméricos são exemplos tipicos de compactação. Estes exemplos servem também para enfatizar a dependência semântica dos métodos de compressão. Note que é necessário ter conhecimento do tipo do dado que está sendo tratado, para que seja possivel a remoção de zeros ou caracteres de espaço, sem prejudicar a informação contida nos dados.

Outro esquema bastante usado é a compactação de chaves com sequências finais redundantes (rear-compaction) [REGH 81]. As chaves de um arquivo estão normalmente ordenadas e alguns bits de informação podem ser retirados sem contudo, prejudicar a ordenação das mesmas. O conjunto de chaves da figura II-1 ilustra tal situação.

informação original		informação compactada		
chave	valor	chave	valor	valor
1	1010101010	1	101010	1010101
2	1010110001	2	1010110	1010110
3	1010111101	3	1010111	1010111
			(a)	(b)

figura II-1 : rear-compaction

As chaves compactadas (coluna (a)) apresentam tamanhos diferentes, o que pode não ser recomendável, entretanto, chaves com tamanhos fixos podem ser sempre conseguidas considerando-se o número de bits da maior chave compactada (coluna (b)).

A não reversibilidade dos dados juntamente com a dependência do contexto da informação, são as principais desvantagens da compactação, contribuindo para o seu pouco uso. Esta tese não aborda as técnicas de compactação de dados.

2.2 Técnicas de compressão

As técnicas de compressão utilizam mecanismos de codificação para reduzir o tamanho dos dados. Esta codificação pode ser entendida como um mapeamento de um conjunto de símbolos fonte, utilizado para representar a informação original, em um conjunto alternativo de símbolos, utilizado para representar a informação comprimida. O processo de descompressão utiliza o mesmo mapeamento, agora no sentido inverso, a fim de obter a informação original através da informação comprimida.

Um conhecimento das caracteristicas dos dados a serem comprimidos é útil para a determinação de uma codificação ideal para uma determinada massa de dados. Análises estatisticas podem ser uma ferramenta importante para obtenção de caracteristicas importantes dos dados. Entretanto, a obtenção destas informações nem sempre é possivel e/ou viável.

Métodos de compressão que consideram as caracteristicas dos dados para a determinação do modo como será feito a codificação têm uma aplicação mais restrita, já que dados que não possuam as caracteristicas assumidas no projeto, não serão comprimidos eficientemente se sujeitos a este método. Algumas destas técnicas são inclusive dependentes da semântica dos dados, o que as torna ainda mais restritas. As referências [LAES 86], [RUTH 72] e [REGH 81] abordam métodos com estas caracteristicas.

Existem outros métodos de compressão que não assùmem nenhum conhecimento a priori dos dados a serem comprimidos. Estes métodos geralmente possuem um mecanismo de aprendizagem das caracteristicas dos dados, o qual é realizado à medida que estes vão sendo comprimidos. Neste caso, qualquer tipo de informação pode ser comprimida. A quantidade de redução obtida com estes métodos depende basicamente de quão redundante são os dados submetidos à compressão.

Em uma primeira análise, um método de compressão deve definir dois conjuntos de simbolos: um para representar os dados originais e outro para representar os dados comprimidos. A forma como é feito o mapeamento entre estes dois conjuntos define o

tipo de código usado. Os códigos podem ser bloco-bloco, variável-variável, bloco-variável ou variável-bloco.

Códigos do tipo bloco-bloco utilizam conjuntos de símbolos de tamanho fixo e definido. Os códigos ASCII, EBCDIC, são exemplos de códigos do tipo bloco-bloco. Este tipo de código não é usado para fins de compressão, uma vez que o mapeamento um a um de símbolos de mesmo tamanho não permite que haja compressão.

Os tipos de códigos restantes podem ser utilizados para compressão. Códigos variável-variável mapeiam sequências de símbolos do conjunto fonte de tamanho variável em sequências do conjunto alternativo também de tamanho variável. Os códigos bloco-variável mapeiam sequências de símbolos do conjunto fonte de tamanho fixo em sequências de símbolos do conjunto alternativo de tamanho variável, já os códigos variável-bloco mapeiam sequências de símbolos do conjunto fonte de tamanho variável em sequências de símbolos do conjunto alternativo de tamanho fixo.

2.2.1 Ganhos e perdas associados à compressão

O benefício mais óbvio da utilização de compressão é a redução do tamanho dos dados armazenados, implicando em um efetivo aumento da capacidade dos meios de armazenamento e/ou diminuição dos custos com transmissão. Entretanto, outros benefícios menos óbvios podem ser destacados.

Uma vez que a informação comprimida representa uma codificação da informação original, a compressão provê, como efeito colateral, uma certa proteção aos dados comprimidos, já que é

necessário saber o código utilizado na compressão para se ter acesso à informação original.

Em sistemas onde a carga de operações de entrada e saida é considerável (I/O bound), a redução do tamanho dos dados manipulados, pode indiretamente aumentar a performance do mesmo, considerando que os canais de comunicação responsáveis pelas operações de entrada e saida apresentarão uma maior vazão, sendo portanto melhor utilizados.

Em bancos de dados, por exemplo, a utilização de compressão em chaves de arquivos permite que mais chaves sejam armazenadas em um bloco de disco, possibilitando uma redução no tempo necessário para pesquisar uma chave e contribuindo para a melhoria do tempo de resposta do sistema. Técnicas de compressão tem sido usadas já a algum tempo neste tipo de sistemas, sendo o VSAM (Virtual Storage Access Method) da IBM um exemplo clássico.

O "overhead" computacional envolvido nos procedimentos de compressão e descompressão é apontado como principal problema na uitilização de compressão de dados para solucionar os problemas de armazenamento e transmissão de dados. Ou seja, deve-se negociar um aumento na capacidade de armazenamento e/ou transmissão em detrimento da quantidade de processamento gasta neste processo.

O intuitivo "trade-off" espaço/tempo discutido acima pode contudo, em certas ocasiões, não ser verdadeiro. Hamaker em [HAMA 86] apresenta uma série de observações que merecem ser consideradas.

Como já apresentado, a compressão é responsável indiretamente por ganhos computacionais de difícil percepção e mensuração. A compressão pode levar a uma redução nas operações físicas de entrada e saída suportadas pela CPU. Esta redução também proporciona uma maior utilização da CPU pelos processos do usuário, uma vez que as operações de entrada e saída sendo menos frequentes, proporcionam uma maior utilização da fatia de tempo num sistema de tempo compartilhado. Este fato contribui ainda para a diminuição do "overhead" de troca de processos e escalonamentos de "jobs".

Qualquer aplicação que manipule uma massa de dados sem observar sua estrutura interna será sempre executada mais rapidamente quando os dados em questão estiverem comprimidos. Operações de "backup/recovery" são exemplos típicos de tais aplicações.

Além do mais, o surgimento de terminais inteligentes possibilita a migração do processamento de descompressão do computador hospedeiro para o terminal. Algumas aplicações onde a descompressão é mais frequente que a compressão (consulta à manuais "on-line", por exemplo), podem se valer deste mecanismo para obter um melhor desempenho.

É necessário se ter em mente que os custos inerentes ao processo de compressão estão sempre associados a ganhos em operações de entrada e saída portanto, uma análise global deve ponderar todos estes fatores, a fim de determinar o custo real envolvido em uma determinada aplicação. Esta não é uma tarefa

fácil, pois se os custos com compressão são óbvios, os ganhos em processamento não o são.

Existem outros problemas associados à compressão de dados, os quais descreveremos a seguir.

O tamanho imprevisivel da informação comprimida apresenta restrições no caso de aplicações que necessitem receber uma taxa constante de informação ou no caso de compressão de registros de um arquivo. Dificuldades na alocação de memória podem surgir também por causa desta caracteristica.

A redução da redundância da informação torna esta mais susceptivel a erros. O prejuizo causado pela ocorrência de um erro em uma informação comprimida depende principalmente se o método é estático ou dinâmico (estes conceitos serão definidos mais adiante). Os erros ocorridos na transmissão geram uma perda de sincronização entre o compressor e o descompressor, de forma que a informação descomprimida não corresponde à informação originalmente comprimida. Nos métodos estáticos a resincronização é naturalmente conseguida no decorrer do processo, entretanto nos métodos dinâmicos a ocorrência de erros pode ser catastrófica, pois a resincronização é muito mais dificil. Deve-se ter muita atenção na escolha de um método de compressão que será empregado para transmitir informação em um canal susceptivel a erros.

Outro problema com compressão, é o menor grau de portabilidade apresentado pela informação comprimida. Para que a informação comprimida em um sistema possa ser descomprimida com sucesso em outro sistema, é necessário que os sistemas concordem no

fácil, pois se os custos com compressão são óbvios, os ganhos em processamento não o são.

Existem outros problemas associados à compressão de dados, os quais descreveremos a seguir.

O tamanho imprevisível da informação comprimida apresenta restrições no caso de aplicações que necessitem receber uma taxa constante de informação ou no caso de compressão de registros de um arquivo. Dificuldades na alocação de memória podem surgir também por causa desta característica.

A redução da redundância da informação torna esta mais susceptível a erros. O prejuízo causado pela ocorrência de um erro em uma informação comprimida depende principalmente se o método é estático ou dinâmico (estes conceitos serão definidos mais adiante). Os erros ocorridos na transmissão geram uma perda de sincronização entre o compressor e o descompressor, de forma que a informação descomprimida não corresponde à informação originalmente comprimida. Nos métodos estáticos a resincronização é naturalmente conseguida no decorrer do processo, entretanto nos métodos dinâmicos a ocorrência de erros pode ser catastrófica, pois a resincronização é muito mais difícil. Deve-se ter muita atenção na escolha de um método de compressão que será empregado para transmitir informação em um canal susceptível a erros.

Outro problema com compressão, é o menor grau de portabilidade apresentado pela informação comprimida. Para que a informação comprimida em um sistema possa ser descomprimida com sucesso em outro sistema, é necessário que os sistemas concordem no

método utilizado. A inexistência de padrões na área de compressão de dados configura-se em um entrave para esta concordância. Outro problema ligado a portabilidade diz respeito à própria implementação do algoritmo. Para obter uma performance aceitável, muitas vezes os algoritmos são codificados em assembler.

Alguns dados podem ter importantes propriedades corrompidas após a compressão. No caso de chaves de um arquivo, a perda da ordenação acarreta um grave problema de desempenho do sistema. O acesso a um registro neste caso, só seria possivel se todas as chaves fossem descomprimidas. Existem técnicas entretanto, que permitem a compressão de chaves sem perda de ordenação, e neste caso, as chaves não precisam ser descomprimidas para que uma busca seja efetuada.

2.2.2 Classificação dos métodos de compressão

Os métodos de compressão podem ser classificados em estáticos e dinâmicos. Nos métodos estáticos, o mapeamento é definido antes do processo de compressão ser iniciado e é mantido sem modificações durante todo o processo. Dessa forma, uma determinada sequência de simbolos do código fonte será sempre mapeada para a mesma sequência de códigos alternativos correspondentes. Alguns algoritmos estáticos realizam o processo de compressão em vários passos (várias leituras dos dados a serem comprimidos).

Nos métodos dinâmicos, o mapeamento é definido durante o processo de compressão. Os métodos dinâmicos são também denominados de adaptativos, e realizam o processo de compressão em um

único passo.

Existem métodos que englobam caracteristicas dos algoritmos estáticos e dinâmicos; são portanto métodos hibridos.

LZW-MODIFICADO é um método de compressão dinâmico.

Os métodos de compressão podem também ser classificados pelo tipo de código que é utilizado no algoritmo (bloco-variável, variável-bloco, variável-variável). Uma outra classificação comum é a divisão dos métodos de compressão em familias de métodos.

A definição de uma familia de métodos depende exclusivamente da solidez dos conceitos de um método precursor. Futuras contribuições e melhoramentos a este método darão sustentação à familia em questão. Obviamente se um determinado método fomenta muitas sugestões, é um indicativo que sua familia se manterá. As seguintes familias serão discutidas neste trabalho: códigos de Huffman, códigos aritméticos e códigos DPM/L. Alguns métodos importantes que não se encaixam nestas familias serão abordados em uma seção à parte.

2.2.2.1 Familia dos códigos de Huffman

Huffman em seu clássico artigo publicado em 1952 [HUFF 52] apresentou um dos primeiros métodos de compressão de dados. O algoritmo de Huffman é bastante importante, não só pelo seu pioneirismo, mas também pela sua simplicidade e eficiência. A importância do algoritmo de Huffman é comprovada pelos inúmeros artigos que o sucederam, e pela frequente utilização de seus percentuais de compressão em "benchmarks" comparativos com outros

métodos.

O algoritmo proposto por Huffman em um primeiro passo, verifica a distribuição dos símbolos do código fonte nos dados a serem comprimidos. A seguir, as distribuições obtidas são ordenadas em uma lista, e uma árvore binária "cheia" (full binary tree) é construida, seguindo o algoritmo apresentado a seguir.

a) cria-se um novo elemento que será, na árvore sendo criada, o pai dos dois elementos da lista com menor distribuição. O ramo da direita é rotulado com o valor 1 enquanto que o ramo da esquerda é rotulado com o valor 0 (ou vice-versa).

b) os elementos escolhidos da lista devem ser retirados e o novo elemento criado no passo a) deve ser incluido na lista com o valor da soma das distribuições dos elementos retirados.

c) se o novo conjunto possuir mais de um elemento retorna-se ao passo a). Caso contrário, a árvore está montada e pode-se parar o algoritmo.

Os códigos gerados por este algoritmo são códigos de redundância minima.

Cada caminho que liga a raiz da árvore a uma folha, define o código associado ao símbolo fonte correspondente àquela folha. A figura II-2 mostra um exemplo da utilização do algoritmo apresentado acima para um código fonte composto de oito símbolos com as

distribuições indicadas.

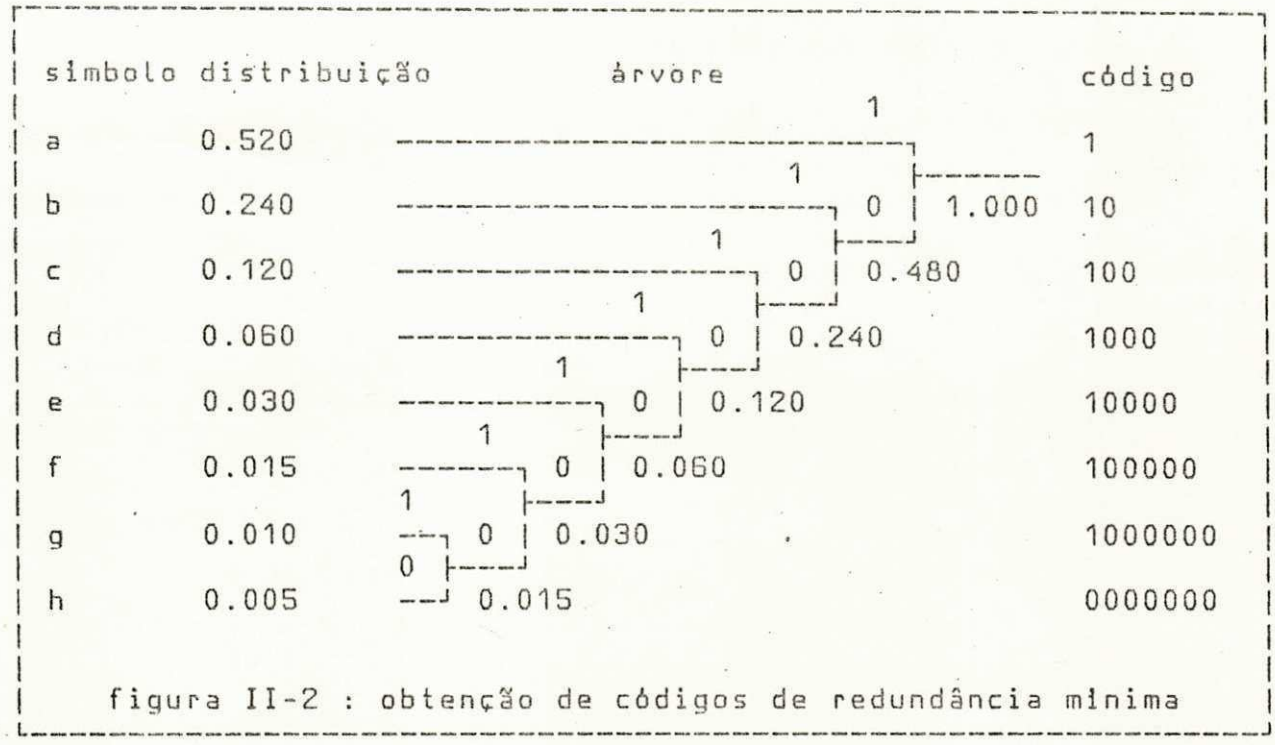


figura II-2 : obtenção de códigos de redundância mínima

O segundo passo do algoritmo de Huffman processa novamente os dados, substituindo cada símbolo fonte pelo seu código correspondente.

Quando vários elementos da lista de distribuição dos símbolos apresentam o mesmo valor, a escolha dos elementos que devem ser retirados da lista, é feita de forma aleatória. Portanto mais de uma árvore pode ser gerada para uma determinada distribuição de símbolos em uma massa de dados. O tamanho dos códigos gerados também pode ser diferente para uma mesma distribuição. Um alfabeto fonte de 5 símbolos, por exemplo, com distribuição { 0.4, 0.3, 0.1, 0.1, 0.1 } poderia ter códigos com tamanhos { 1, 2, 3, 3, 3 } ou { 1, 2, 2, 3, 4 }, ambos seriam códigos de redundância mínima.

Os códigos obtidos pelo primeiro passo do algoritmo são usados durante todo o processo de compressão realizado no segundo passo. Este é portanto, um método estático.

Gallager em [GALL 78] apresenta alguns resultados e introduz uma série de modificações ao algoritmo originalmente proposto por Huffman. Entre os resultados apresentados por Gallager está a determinação de um limite máximo de redundância dos códigos de Huffman. Neste mesmo artigo, Gallager propõe uma implementação de um método dinâmico baseado nos conceitos do código de Huffman. Este método é executado em um único passo.

A execução em dois passos do método de Huffman compromete bastante a velocidade de compressão do mesmo. Um conhecimento a priori do tipo de informação a ser comprimida poderia evitar o primeiro passo do algoritmo. McIntyre em [MCIN 85] apresenta resultados de compressão de arquivos contendo programas fonte em várias linguagens utilizando códigos de Huffman previamente determinados. Os resultados obtidos indicam que a utilização de tabelas estáticas previamente definidas, além de aumentar a velocidade de compressão do algoritmo (pois elimina o primeiro passo do mesmo), consegue percentuais de compressão próximos do atingido pelo algoritmo original e melhores que o atingido por algoritmos que utilizam uma tabela dinâmica.

Outras referências dão contribuições ao algoritmo proposto por Huffman. Exemplos: [TANA 87] propõe um implementação em hardware; [FALL 73] e [VITT 87] propõem códigos de Huffman adaptativos.

2.2.2.2 Familia dos códigos aritméticos

Os métodos desta familia realizam os processos de compressão e descompressão de uma forma diferente dos demais métodos de compressão. Nestes métodos não existe um mapeamento puro e simples dos simbolos do código fonte nos simbolos do código alternativo. A codificação da mensagem é feita através de operações aritméticas sobre os simbolos componentes da mensagem. A descompressão realiza operações análogas para obter a mensagem original.

Shannon em [SHAN 49] apresentou o método precursor da familia dos códigos aritméticos. Segundo o algoritmo de Shannon, para comprimir uma mensagem de tamanho N, primeiro era necessário ordenar as mensagens pelas suas probabilidades e associar, na ordem, cada mensagem à probabilidade acumulada das mensagens precedentes. O processo descompressor receberia a mensagem codificada como uma fração binária, que seria decodificada por comparação de magnitude.

A partir das idéias de Shannon, Elias sugeriu o primeiro código aritmético em um artigo não publicado. Em 1963 Abramson publicou as sugestões de Elias [ABRA 63]. Elias notou que o esquema proposto por Shannon, funcionaria mesmo que as mensagens não fossem ordenadas, e a probabilidade cumulativa de uma mensagem de N simbolos poderia ser calculada recursivamente através da probabilidade de um simbolo e da probabilidade acumulada da mensagem de N - 1 simbolos.

Os códigos aritméticos apresentaram inicialmente uma série de deficiências, entretanto muito foi feito pra o seu desenvolvimento. Os artigos [PASC 76], [RISS 76], [RUBI 79], [LANG 84] e [WITT 87] apresentam uma série de sugestões e refinamentos às idéias de Elias.

Os dados comprimidos através de um código aritmético são representados por um número real dentro do intervalo [0,1). Inicialmente o intervalo [0,1) é subdividido em intervalos que correspondem ao valor da probabilidade acumulada de cada simbolo do código fonte (figura II-3 (a)).

Cada simbolo processado será responsável por um encurtamento do intervalo correntemente usado. Quanto menor o intervalo, mais bits serão necessários para representá-lo. A presença de um simbolo com probabilidade alta na massa de dados a ser comprimida, corresponde a um pequeno encurtamento no intervalo, enquanto que a presença de um simbolo com probabilidade baixa corresponde a um encurtamento maior.

O exemplo da figura II-3 tornará mais claro o processo de compressão através de um código aritmético. Qualquer valor no intervalo [0.2536, 0.2552), pode ser usado para representar a informação comprimida. Na figura II-4 o processo de descompressão do exemplo da figura II-3 é ilustrado, utilizando o valor 0.2536 para representar a informação comprimida.

símbolos	distribuição
a	0.2
b	0.4
c	0.1
d	0.2
e	0.1

(a) intervalo inicial

(b) compressão da mensagem bacd

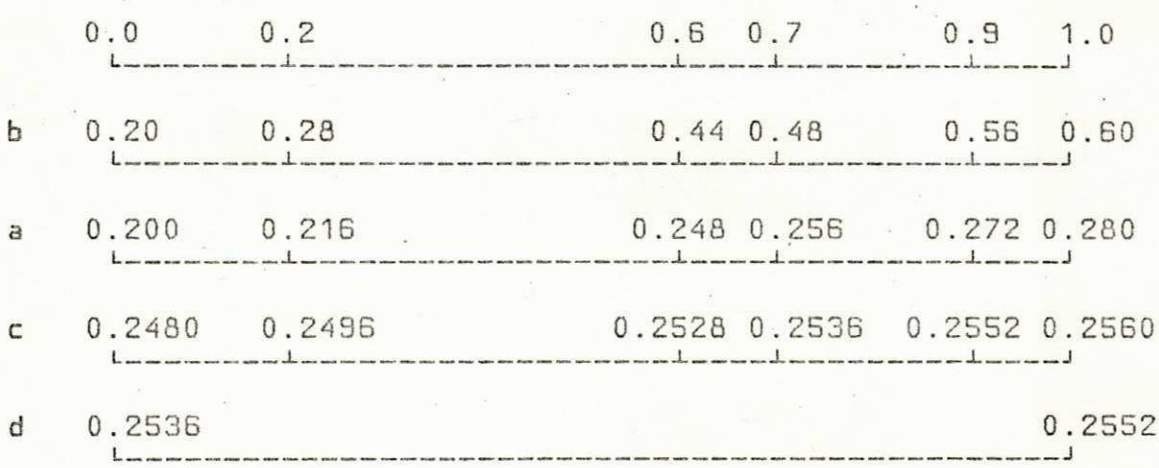


figura II-3 : exemplo de compressão com códigos aritméticos

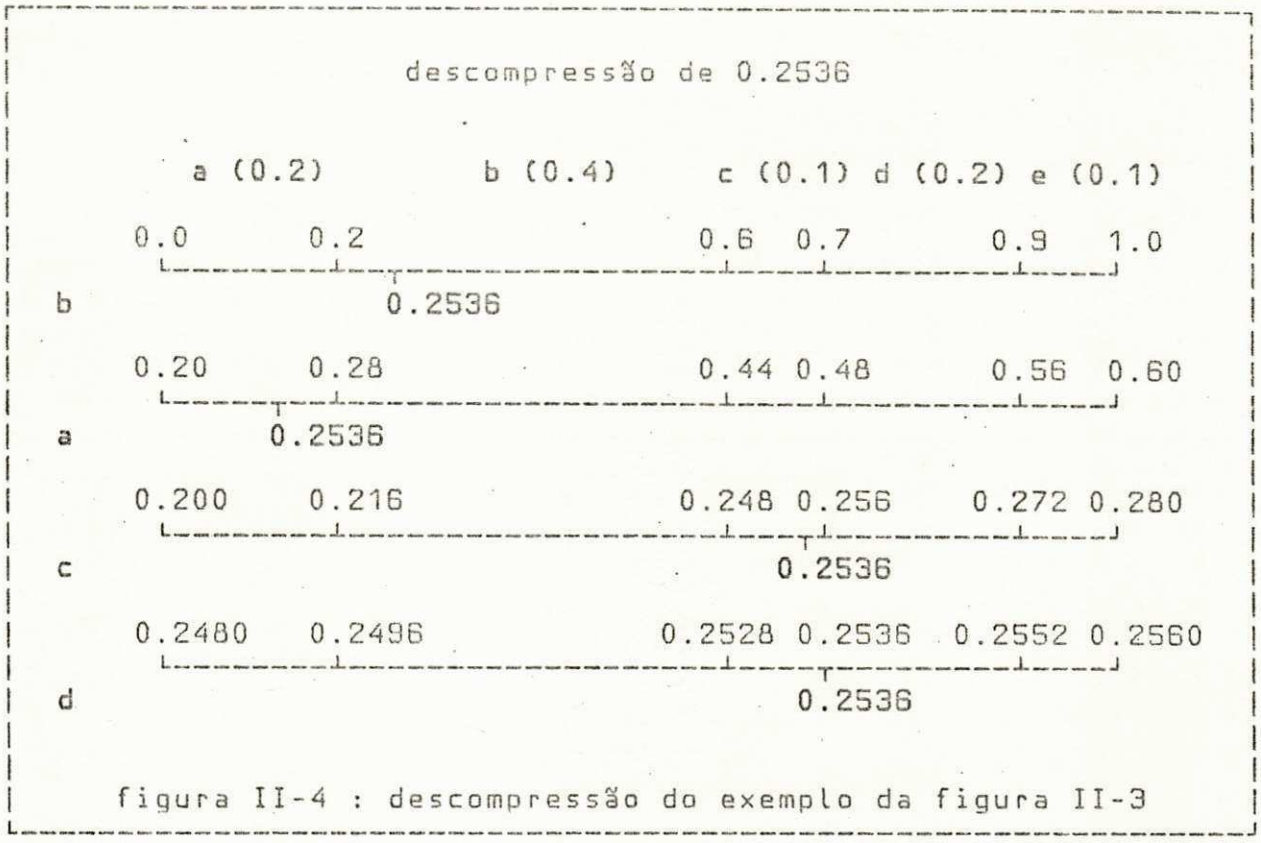


figura II-4 : descompressão do exemplo da figura II-3

O tamanho do último subintervalo calculado no processo de compressão indica o número de bits necessários para a representação da informação comprimida. O tamanho do intervalo final, como podemos perceber à partir do exemplo da figura II-3, é igual ao produtório das probabilidades dos simbolos, que por sua vez representa a entropia da mensagem. Portanto a informação comprimida é representada por um número minimo de bits, obtendo uma compressão máxima.

O percentual de compressão é entretanto diminuido, pela necessidade do descompressor de saber das distribuições de probabilidades dos simbolos do código fonte, a fim de conseguir recuperar a informação original com êxito. Além disso, alguns outros problemas também devem ser resolvidos para viabilizar a

utilização de códigos aritméticos. Em geral as soluções destes problemas concorrem para uma pequena redução do percentual de compressão do método.

Um desses problemas diz respeito ao ponto de parada do processo de descompressão. Uma solução para este problema é mandar juntamente com a distribuição de probabilidades e o intervalo a ser decodificado, o tamanho original dos dados. Esta solução apresenta como principal desvantagem a necessidade da execução de um passo extra sobre os dados para determinar o seu tamanho ou a necessidade da utilização de buffers. Uma segunda solução é a adoção de um simbolo terminal (EOF, por exemplo) que deveria sempre aparecer no final dos dados a serem comprimidos.

Outro problema bastante discutido é a necessidade de uma grande precisão nas operações realizadas sobre os intervalos. O gerenciamento de "underflow" e "overflow" deve ser realizado com bastante cuidado. A corrupção de um bit pode causar enormes modificações nos dados comprimidos.

2.2.2.3 Familia dos códigos OPM/L

Storer em [STOR 82], apresenta um modelo geral de técnicas de compressão que encaixa uma grande variedade de métodos apresentados em outras literaturas, como casos especiais deste modelo.

A idéia básica é substituir sequências de símbolos do código fonte que já foram vistas "em um passado recente" por apontadores para estas sequências, ou seja, se uma determinada sequência de

símbolos aparece mais de uma vez em uma massa de dados basta copiá-la a primeira vez, e nas vezes subsequêntes indicar sua presença através de apontadores. Os apontadores representam o inicio da representação da sequência de símbolos e o seu tamanho. Estes métodos são adaptativos e apresentam um excelente percentual de compressão. Os métodos OPM/L correspondem a um dos ramos definidos por Storer. Diversos trabalhos publicados, consolidaram a importância destes métodos.

Os métodos até aqui discutidos geralmente consideram redundâncias de uma forma global dentro de uma massa de dados a ser comprimida. Redundâncias localizadas não foram consideradas. Os métodos da familia OPM/L tentam reduzir este tipo de redundância.

Um dos primeiros métodos desta familia foi proposto po Ziv e Lempel (LZ) em [ZIV 77]. Este método baseia-se na utilização de uma janela com um tamanho fixo que delimita o alcance dos apontadores. A janela contém N símbolos e é continuamente atualizada à medida que desliza sobre a informação sendo comprimida.

O tamanho fixo da janela possibilita o uso de apontadores com tamanhos fixos, definindo assim um código do tipo variável-bloco que mapeia uma sequência com um número qualquer de símbolos contidos na janela em um apontador com tamanho fixo igual a log N bits. A janela é responsável por manter um dicionário de símbolos que representam com uma probabilidade muito alta um contexto comum.

O artigo de Ziv e Lempel se preocupou principalmente com as

bases teóricas do algoritmo LZ, sem se preocupar muito com os aspectos de implementação. No ano seguinte, um outro artigo [ZIV 78] apresentou uma variação do primeiro algoritmo mais rápida e mais fácil de implementar.

Os algoritmos apresentados atingem um alto percentual de compressão. Entretanto, embora a velocidade de descompressão seja alta, o mesmo não é verdade para a velocidade de compressão.

Bell em [BELL 86], apresenta uma otimização para os algoritmos acima, de modo a conseguir uma velocidade dez vezes maior. Contudo a velocidade atingida ainda é muito baixa. Welch em [WELC 84] sugere simplificações no algoritmo LZ que possibilitam velocidades de compressão muito maiores, sem que ocorra uma queda pronunciada no percentual de compressão atingido. Welch propõe um algoritmo (LZW) para implementação em hardware. O capitulo III discute a técnica LZW em detalhes.

Bell apresenta alguns resultados de comparações do desempenho dos métodos apresentados nesta seção com o método de Huffman e um código aritmético. A figura II-5 mostra estes resultados.

compressão de um arquivo contendo texto com 139.521 bytes

método	LZ	LZW	aritmético	Huffman
% compres.	49,7%	47,9%	42,6%	58,7%
vel. comp.	24 c/s	5700 c/s	-	990 c/s
vel. desc.	15200 c/s	8400 c/s	-	1300 c/s
memória	8K	48K	32-1400K	8K

(c/s - caracteres por segundo)

todos os algoritmos foram implementados em software

figura II-5 : desempenho de alguns métodos de compressão

2.2.2.4 Outros métodos

Alguns outros métodos de compressão, embora não tenham sido discutidos o bastante a ponto de originar uma familia, destacam-se na literatura e são apresentados nesta seção.

a) Run-length encoding .

A ocorrência de longas sequências de um mesmo simbolo nos dados a serem comprimidos é bastante comum em determinados tipos de informação. O processo de codificação destas sequências (run-length enconding) pode ser feito através de um inteiro, representando o número de simbolos da sequência, seguido do simbolo repetido. Uma sequência de dez simbolos a por exemplo, seria codificada pelos simbolos 10 e a alcançando um percentual de compressão de 80%.

Para diferenciar os simbolos normais dos simbolos que representam sequências de simbolos repetidos é necessário a utilização de um simbolo de escape que será colocado antes da representação de uma sequência de simbolos repetidos. No exemplo acima teríamos: ESCAPE 10 a.

O número de simbolos repetidos que podem ser representados pelo esquema proposto é limitado ao maior inteiro que pode ser representado em um simbolo (se os simbolos forem caracteres ASCII, este valor é 256). Existe também um limite inferior, ou seja, o número mínimo de simbolos repetidos que compensa a utilização da representação apresentada. É fácil de ver que a utilização deste mecanismo para comprimir sequências com dois simbolos repetidos não é interessante.

A eventual possibilidade do simbolo de escape poder aparecer como um simbolo dos dados, exige mecanismos para prover a transparência deste simbolo nos dados. A utilização do simbolo de escape e a inclusão de mecanismos para prover transparência, reduz um pouco o percentual de compressão obtido.

Este tipo de compressão é bastante usado e apresenta como principais vantagens sua simplicidade e potencial de compressão em determinados tipos de informação (imagens, por exemplo). Por outro lado, alguns tipos de informação quase não apresentam longas sequências de simbolos repetidos e portanto o uso desta técnica não é recomendável. Muitas técnicas utilizam o método apresentado em um filtro colocado na entrada dos dados. Esse mecanismo permite a obtenção de ganhos consideráveis no percen-

tual de compressão. A seção 5.5 do capitulo 5 analisa a utilização de um filtro deste tipo, em associação com o algoritmo LZW-MODIFICADO.

b) Compressão de dados ordenados

Este método baseia-se num processo de diferenciação entre entradas (registros) consecutivas de um arquivo. Quando estas entradas têm informação comum, apenas a informação diferente precisa ser realmente armazenada. A informação comum só precisa ser armazenada em uma entrada, bastando que exista na outra entrada uma referência a esta informação.

O exemplo da figura II-6 apresenta uma lista de nomes ordenados e uma lista de nomes comprimidos. Cada elemento da lista comprimida é representado pelo número de simbolos comuns a este e ao nome anterior e pelos simbolos diferentes. O processo de descompressão desse tipo de representação geralmente é bastante lento. Para descomprimir qualquer nome é necessario uma leitura sequencial dos nomes anteriores. Entretanto, em algumas aplicações esta técnica pode ser útil.

nome	nome comprimido
abel	abel, 0
abelardo	ardo, 4
basilio	basilio, 0
bastos	tos, 3
carlos	carlos, 0
carlitos	itos, 4

figura II-6 : compressão de dados ordenados

c) Dicionários de palavras

Estes métodos utilizam uma entidade auxiliar denominada dicionário. O dicionário geralmente é formado por sequências de símbolos (palavras) que têm grande probabilidade de estar presentes nos dados a serem comprimidos. A obtenção das palavras que devem constar no dicionário é feita através de uma análise estatística da informação a ser comprimida.

A informação comprimida consiste de referências ao dicionário e símbolos do alfabeto fonte. As palavras nos dados a serem comprimidos que não aparecem no dicionário são copiadas nos dados comprimidos sem alteração; é necessário portanto, mecanismos que permitam a diferenciação entre este tipo de informação e as referências ao dicionário.

O processo de descompressão é trivial, e consiste basicamente de consultas ao dicionário.

d) Dupla compressão

Como foi visto anteriormente, na compactação, a redundância eliminada depende fortemente da semântica dos dados. A redundância independente da semântica dos dados não é reduzida. Assim, uma massa de dados sujeita à compactação pode sofrer um futuro processo de compressão, obtendo uma redução ainda maior. Por outro lado, a utilização de dupla compressão, ou seja, compressão de uma massa de dados que já foi sujeita a um processo de compressão, em geral não surte efeito, já que a massa de dados, após a primeira compressão, praticamente não apresenta nenhuma

redundância.

A execução em dois passos, característica dos métodos que utilizam dupla compressão, é responsável pelas baixas velocidades de compressão destes métodos e a não adequação para o uso em comunicação "on-line".

Em alguns casos, a utilização de dupla compressão pode até acarretar um aumento na massa de dados. Entretanto, em circunstâncias especiais, a dupla compressão pode ser realizada. O algoritmo a seguir, apresentado por Bassiouni em [BASS 86], descreve um método de compressão que é realizado em duas etapas e que apresenta uma alta taxa de compressão.

A idéia básica do método é, através da utilização de pacotes de códigos com um formato "bem comportado" no primeiro processo de compressão, prover redundância suficiente, para que um segundo passo sobre a informação comprimida, produza percentuais de compressão compensadores.

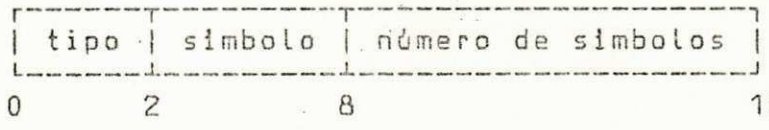
O método utiliza um dicionário com 1024 entradas que pode ser estático ou construído dinamicamente (nesse caso é necessário um passo extra para leitura dos dados a serem comprimidos). O dicionário contém as 1024 palavras mais frequentes nos dados a serem comprimidos e é ordenado em ordem decrescente de frequência das palavras.

No primeiro passo, a compressão se dá segundo um mapeamento da informação original em quadros de códigos com formatos previamente definidos. Os formatos utilizados são:

- formato dos quadros utilizados para compressão de palavras contidas no dicionário

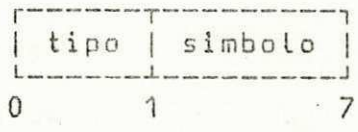
O campo tipo (bits 0 e 1) indica o formato do quadro. Para o formato do quadro acima o tipo tem valor 10. O campo maiúsculo (bit 2) indica se a palavra inicia com letra maiúscula.(bit ligado). O campo terminação (bits 3, 4 e 5) é usado para indicar a terminação da palavra (é comum existirem palavras que apresentam um radical comum e distiguem-se apenas na terminação). Além disso, existem terminações que são bastante frequentes. O método foi desenvolvido para comprimir textos em inglês, e apenas os radicais eram armazenados no dicionário. Sete terminações comuns foram utilizadas (ed, ing, d, s, etc.). Se a palavra não apresentasse nenhuma das terminações utilizadas, o campo terminação teria o valor 000). O campo endereço (bits 6 a 15) constitui o endereço de 10 bits correspondente à posição da palavra no dicionário.

- formato de quadros utilizados para compressão de sequências de caracteres repetidos



O campo tipo (bits 0 e 1) tem a mesma função descrita no quadro anterior. Seu valor para este quadro é 11. O campo simbolo (bits 2 a 7) indica o simbolo que está sendo repetido. O campo número de simbolos (bits 8 a 15) indica o número de simbolos repetidos. Este quadro permite a associação de run-length encoding ao método.

- formato de quadros utilizados para representar simbolos que não formam palavras do dicionário nem sequências de simbolos repetidos



O campo tipo (bit 0) é sempre igual a zero para simbolos não comprimidos. O campo simbolo (bits 1 a 7) representa o simbolo não comprimido (supõe-se um código de 7 bits, como o ASCII).

Os formatos dos quadros descritos acima, foram cuidadosamente projetados para permitir que os dados comprimidos em um primeiro passo, apresentem uma série de caracteristicas que possibilite uma segunda compressão com ganhos realmente consideráveis.

A figura II-7 apresenta uma comparação entre os percentuais de compressão do primeiro passo e do segundo passo da técnica apresentada e de uma implementação de um código de Huffman adaptativo. Para os arquivos utilizados o esquema proposto apresenta um pequeno ganho frente ao código de Huffman no

primeiro passo. O ganho apresentado pelo segundo passo entretanto, é bastante considerável.

arquivo	Huffman	passo um	passo dois
1	44%	46%	67%
2	46%	46%	62%
3	50%	56%	76%
4	58%	58%	78%

arquivo 1 - parte do manual do UNIX (72.523 bytes)
arquivo 2 - manual com figuras (30.904 bytes)
arquivo 3 - texto (48.308 bytes)
arquivo 4 - programas em pascal (58829 bytes)

figura II-7 : comparação de percentuais de compressão
Bassiouni X Huffman

e) Compressão com preservação de ordenação

Estes métodos são utilizados para compressão de indices em bancos de dados. Nestas aplicações, é importante que certas caracteristicas dos dados, como a ordenação, sejam mantidas inalteradas após a compressão.

A associação de um código número a cada um dos indices, de forma que a ordenação numérica dos códigos preserve a ordenação lexicográfica dos indices, é uma maneira bastante comum de resolver este problema.

Quando o dominio dos indices é estático, ou seja, existe um número limitado de valores para os indices, o problema tem solução bastante simples. O arquivo contendo informações dos alunos de uma turma de graduação indexada pelos nomes dos alunos,

pode ter seus índices comprimidos utilizando-se uma tabela de mapeamento que associe cada nome a um número, mantendo-se a ordenação original. Entretanto se for possível a inclusão de um novo aluno, o domínio dos índices não será mais estático, e a compressão dos índices é um pouco mais complicada.

A compressão de índices com domínios dinâmicos exige uma operação de recodificação sempre que um novo índice é incluído. Para evitar a operação de recodificação a cada inclusão, é necessário deixar espaços na tabela de mapeamento para os novos índices a serem incluídos. Um algoritmo bastante usado para associar códigos numéricos aos novos índices incluídos é conhecido como o algoritmo de ordenação verdadeira (true order) [BASS 85]. Este algoritmo associa a um novo índice incluído, um valor que é a média entre o valor associado ao índice antecessor e valor associado ao índice sucessor. Se não existem mais códigos disponíveis é necessário uma recodificação. A figura II-8 ilustra a inclusão de índices em um arquivo com domínio dinâmico, utilizando o algoritmo acima descrito.

(1)		(2)		(3)		(4)		(5)
ana	8	ana	8	ana	8	ana	8	recodificar
cecilia	16	beatriz	12	beatriz	12	beatriz	12	
jorge	24	cecilia	16	carlos	14	bruno	13	
maria	28	jorge	24	cecilia	16	carlos	14	
		maria	28	jorge	24	cecilia	16	
				maria	28	jorge	24	
						maria	28	

(1) - situação inicial, nomes mapeados para inteiros
(2) - situação após a inclusão da chave beatriz
(3) - situação após a inclusão da chave carlos
(4) - situação após a inclusão da chave bruno
(5) - inclusão da chave benicio, por exemplo, causaria recodificação

figura II-8 : exemplo do algoritmo true order

A principal desvantagem dos métodos apresentados nesta seção, é sua utilização limitada. Estes métodos apresentam uma dependência de contexto que embora não seja tão forte quanto a apresentada pelas técnicas de compactação, restringe bastante sua utilização. Este fato concorre para a pouca discusão e desenvolvimento destes métodos, que entretanto podem ser responsáveis por excelentes indices de compressão em determinadas aplicações.

2.3 Sumário

A revisão de métodos de compactação e compressão apresentada neste capitulo, dá uma visão geral do grau de desenvolvimento desta área. As familias de métodos de compressão mais importantes foram discutidas, tendo suas vantagens, desvantagens, limitações, velocidades de compressão e descompressão e percentuais de compressão ilustrados.

Os métodos de compactação, e alguns métodos de compressão dependentes de semântica, apresentam a séria desvantagem de serem dependentes do contexto ao qual os dados estão associados, e não foram aprofundados neste capitulo.

A utilização de dupla compressão, geralmente não concorre para a obtenção de uma diiferença grande, entre o percentual de compressão obtido na primeira compressão e o obtido na segunda compressão. Não obstante, Bassiouni apresentou um método que utiliza dupla compressão, obtendo um excelente percentual de compressão, para arquivos contendo textos. O algoritmo é realizado em no mínimo dois passos (um para cada compressão) e além de ser lento, não é adequado para comunicação "on-line".

Métodos em mais de um passo (Huffman, por exemplo), embora obtenham bons percentuais de compressão, geralmente são lentos e não se prestam à comunicação "on-line".

Códigos aritméticos atingem, teoricamente, um percentual de compressão igual à entropia da mensagem, entretanto, é necessário a introdução de um certo "overhead" para possibilitar a descompressão da mensagem. O "overhead" acrescido faz com que o percentual de compressão diminua.

Os métodos da familia OPM/L têm como principais caracteristicas, altos percentuais de compressão e nenhuma restrição ao tipo de informação a ser comprimida. Os métodos desta familia são adaptativos e são executados em um passo.

O algoritmo LZ, um dos principais representantes desta família, apresenta altos percentuais de compressão, entretanto, a velocidade de compressão é extremamente baixa.

O algoritmo LZW, em troca de uma pequena queda no percentual de compressão, consegue um enorme aumento da velocidade de compressão, comparando-se com o algoritmo LZ. O algoritmo LZW perde, em um determinado ponto, o potencial de se adaptar aos dados sendo comprimidos, fato que acarreta uma diminuição do percentual de compressão atingido.

O algoritmo LZW-MODIFICADO, proposto nesta tese, é também um método da família OPM/L, e resolve o problema de adaptabilidade do algorito LZW. As velocidades de compressão e descompressão não são contudo, afetadas, enquanto que o percentual de compressão é aumentado.

Um sério problema do algoritmo LZW não é resolvido pelo algoritmo LZW-MODIFICADO. A perda de sincronização entre as funções de compressão e descompressão de uma massa de dados, acarreta uma regeneração errada da informação comprimida, a partir do ponto onde a sincronização foi perdida. Este é um problema comum em métodos adaptativos.

O próximo capitulo traz detalhes da especificação do algoritmo LZW-MODIFICADO.

CAPITULO III

Especificação do algoritmo LZW-MODIFICADO

A seção 2.2.2.3 do capitulo anterior apresenta a evolução dos métodos de compressão de dados que formam a familia OPM/L. Um dos métodos lá tratados é o sugerido por Welch em [WELC 84] denominado LZW. O método LZW por sua vez, constitui uma modificação do algoritmo de Lempel-Ziv (LZ) [ZIV 77]. O método LZW pode ser implementado de maneira bastante simplificada , obtendo uma velocidade de compressão muito superior a do seu predecessor. A simplicidade da implementação permite o desenvolvimento de projetos a nivel de hardware.

A pequena queda no percentual de compressão do algoritmo LZW frente ao algoritmo LZ, decorrente das simplificações impostas, é desprezivel em relação ao aumento da velocidade alcançado. A figura III-1 apresenta um quadro comparativo do percentual de compressão e das velocidades de compressão e descompressão de duas implementações em software dos algoritmos LZ e LZW, executadas em um VAX 11/750 [BELL 86].

método	percentual compressão	velocidade compressão	velocidade descompressão
LZ	49,7%	24 c/s	15.200 c/s
LZW	47.9%	5700 c/s	8.400 c/s

figura III-1 : comparação dos métodos LZ e LZW

Embora a amostra de dados utilizada nesta análise não seja muito representativa (foi utilizado apenas um arquivo contendo texto em inglês de tamanho 139.521 bytes), pode-se perceber que o ganho de 23.650% obtido pela técnica LZW na velocidade de compressão é muito grande, enquanto que a perda no percentual de compressão de apenas 3,7% representa muito pouco.

Os algoritmos da familia OPM/L são adaptativos. O algoritmo LZW entretanto, perde em um determinado ponto, o poder de se adaptar aos dados sendo comprimidos. Este fato contribui para uma queda do percentual de compressão atingido em certos tipos de arquivos. Sugere-se uma modificação ao algoritmo LZW, de modo que este possa sempre se adaptar às novas redundâncias contidas nos dados a serem comprimidos, obtendo em média, um maior percentual de compressão. As próximas seções detalharão o algoritmo LZW e o algoritmo LZW-MODIFICADO que engloba as modificação sugeridas.

3.1 Algoritmo LZW

3.1.1 Compressão

O algoritmo de compressão de dados LZW, baseia-se na manipulação de uma Tabela de Simbolos (TS) que é dinâmicamente montada à medida que a informação a ser comprimida é lida. Esta tabela é a responsável pelo mapeamento de uma sequência de caracteres de comprimento aleatório em um código com um número constante de bits. Este código representa a posição da TS onde está armazenada a sequência de caracteres a ser codificada, e seu tamanho é log N, onde N é o tamanho da TS.

Cada simbolo presente na TS é formado por um prefixo e um caractere de extensão. O prefixo por sua vez, é um simbolo que já pertence à TS, ou seja, se o simbolo wK formado pelo prefixo w e pelo caractere de extensão K pertence à TS, então o prefixo w é um simbolo que também pertence à TS.

A TS inicialmente contém todos os caracteres do código fonte, ou seja, as 256 primeiras entradas da TS estão ocupadas com os 256 caracteres componentes do código fonte (256 combinações possiveis para um código de 8 bits como o ASCII por exemplo).

Na geração dos códigos comprimidos, a técnica LZW usa um algoritmo "avarento", que recebe os caracteres da entrada um a um, tentando reconhecer um simbolo com a maior sequência de caracteres que já existe na TS. Quando finalmente se encontra um sequência que não pertence à TS, esta é incluida na TS e o código associado à última sequência (simbolo) reconhecida é emitido. Em seguida o processo é repetido utilizando como prefixo o último caratere lido. A figura III-2 apresenta o algoritmo de compressão LZW.

```
        Inicialize TS com todos os simbolos de um caractere
        Leia o primeiro caractere do arquivo a ser comprimido
        Atribua caractere a prefixo
laço:   Leia próximo caractere do arquivo a ser comprimido
        Se não existem mais caracteres
          Emita o codigo associado a prefixo
          Fim
        Senão
          Se existe prefixo+caractere em TS
            Atribua prefixo+caractere a prefixo
          Senão
            Emita o codigo associado a prefixo
            Inclua prefixo+caractere em TS
            Atribua caractere a prefixo
            Vá para laço
```

figura III-2 : algoritmo de compressão da técnica LZW

O exemplo da figura III-3 servirá para esclarecer melhor a operação de compressão LZW. Neste exemplo os dados são representados apenas pelos caracteres a, b e c. A compressão da mensagem ababba, seguiria os passos lá indicados.

```
Inicilaizar TS com todos os caracteres (1-a, 2-b, 3-c)
Lê o primeiro caractere (a)
Atribui (a) a prefixo
Lê o próximo caractere (b)
Verifica se (a)(b) está em TS
Como (a)(b) não está em TS então
  Emite (1) (código associado a (a))
  Inclui (a)(b) em TS (1-a, 2-b, 3-c, 4-ab)
  Atribui (b) a prefixo
Lê o próximo caractere (a)
Verifica se (b)(a) está em TS
Como (b)(a) não está em TS então
  Emite (2) (código associado a (b))
  Inclui (b)(a) em TS (1-a, 2-b, 3-c, 4-ab, 5-ba)
  Atribui (a) a prefixo
Lê o próximo caractere (b)
Verifica se (a)(b) está em TS
Como (a)(b) está em TS então
  Atribui (a)(b) a prefixo
Lê o próximo caractere (b)
Verifica se (ab)(b) está em TS
Como (ab)(b) não está em TS então
  Emite (4) (código associado a (ab))
  Inclui (ab)(b) em TS (1-a, 2-b, 3-c, 4-ab, 5-ba, 6-abb)
  Atribui (b) a prefixo
Lê o próximo caractere (a)
Verifica se (b)(a) está em TS
Como (b)(a) está em TS então
  Atribui (b)(a) a prefixo
Lê o próximo caractere (EOF)
  Emite (5) (código associado a (ba))
Fim
```

figura III-3 : exemplo de compressão da mensagem ababba

Admitindo-se 8 bits por caractere e 10 bits por código comprimido (TS com 1024 entradas), o conteúdo original do arquivo tem 48 bits, enquanto que o conteúdo comprimido resulta em apenas 40 bits.

O tamanho variável das sequências correspondentes a cada uma das entradas da TS (no exemplo da figura III-3, a configuração final da TS apresenta as entradas 1, 2 e 3 com tamanho 1, as

entradas 4 e 5 com tamanho 2 e a entrada 6 com tamanho 3) complica a implementação do algoritmo, quer seja esta feita em hardware ou em software. Uma melhor representação de TS seria armazenar cada simbolo de TS na forma de um código consistindo de um prefixo mais um caractere de extensão. A TS do exemplo da figura III-3 teria a seguinte configuração: (1-0a, 2-0b, 3-0c, 4-1b, 5-2a, 6-4b), onde cada entrada de TS tem um tamanho fixo. Esta será a representação usada daqui em diante.

3.1.2 Descompressão

A descompressão LZW constrói, à medida que os códigos comprimidos vão sendo interpretados e a informação original vai sendo restaurada, a mesma TS gerada pela compressão. Cada código recebido é desmembrado através de uma consulta à TS em um prefixo e um caractere de extensão. O prefixo encontrado é também desmembrado em um outro prefixo e um outro caractere de extensão, recursivamente até que se chegue a um prefixo com um único caractere.

Cada código recebido (exceto o primeiro) gera uma atualização na TS. O novo código que é acrescentado à TS é formado pelo código que antecedeu o que está sendo decodificado (que será o prefixo) e o último caractere obtido desta decodificação (que será o caractere de extensão). Note que a sequência de caracteres obtida na decodificação se encontra invertida, portanto o último caractere decodificado é na realidade o primeiro caractere lido na compressão. A figura III-4 apresenta o algoritmo de descompressão da técnica LZW.

```
        Inicialize TS com todos os simbolos de um caractere
        Leia o primeiro código do arquivo a ser descomprimido
        Atribua código a código_anterior
        Emita o caractere associado ao código lido
codigo: Leia o próximo código do arquivo a ser descomprimido
        Atribua código a codigo_entrada
        Se não existem mais códigos
          Fim
        Senão
simbolo: Se código não está associado a um único caractere
           Emita o caractere de extensão associado a código
           Atribua o prefixo associado a código a código
           Vá para simbolo
         Senão
           Emita o caractere associado a código
           Inclua código_antigo+caractere em TS
           Atribua código_entrada a código_antigo
           Vá para código
```

figura III-4 : algoritmo de descompressão da técnica LZW

A figura III-5 indica os passos que devem ser seguidos para descomprimir os códigos 1 2 4 e 5 gerados pelo exemplo da figura III-3.

```
Inicializar TS com todos os caracteres (1-0a, 2-0b, 3-0c)
Lê o primeiro código (1)
Atribui código a código_antigo
Emite (a) (simbolo associado a (1))
Lê o próximo código (2)
Atribui código a código_entrada
Verifica se (2) está associado a um único caractere
Como está, então
     Emite (b) (caractere associado a (2))
     Inclua código_antigo+caractere em TS (4-1b)
     Atribua código_entrada a código_antigo
Lê o próximo código (4)
Atribui código a código_entrada
Verifica se (4) está associado a um único caractere
Como não está, então
     Emite (b) (caractere de extensão associado a (4))
     Atribui (1) a código (prefixo associado a (4))
Verifica se (1) está associado a um único caractere
Como está, então
     Emite (a) (caractere associado a (1))
     Inclua código_antigo+caractere em TS (5-2a)
     Atribua código_entrada a código_antigo
Lê o próximo código (5)
Atribui código a código_entrada
Verifica se (5) está associado a um único caractere
Como não está, então
     Emite (a) (caractere de extensão associado a (5))
     Atribui (2) a código (prefixo associado a (5))
Verifica se (2) está associado a um único caractere
Como está, então
     Emite (b) (caractere associado a (2))
     Inclua código_antigo+caractere em TS (6-4b)
     Atribua código_entrada a código_antigo
Lê o próximo código (EOF)
Fim
```

figura III-5 : exemplo de descompressão dos códigos 1 2 4 5

O exemplo III-5 deixa claro o problema da geração invertida da informação descomprimida mencionado anteriormente. A mensagem obtida após a descompressão no exemplo III-5 é abbaab e não a mensagem original ababba. O processamento recursivo de cada código a ser descomprimido é o responsável por esta inversão. Este problema entretanto, pode ser facilmente resolvido através

da introdução de uma estrutura do tipo LIFO (Last In First Out) para armazenar temporariamente os caracteres gerados na descompressão de cada código.

3.2 Algoritmo LZW-MODIFICADO

A tabela de símbolos (TS) da técnica LZW tem tamanho limitado a 2^N símbolos onde N é o número de bits nos códigos comprimidos. É possível pois, no processo de compressão, gerarmos novos símbolos que não possam ser incluídos na TS, por falta de códigos.

Os símbolos encontrados após o total preenchimento da TS serão descartados, ou equivalentemente, as redundâncias que aparecerem à partir deste ponto serão desperdiçadas, concorrendo para que o percentual de compressão diminua (ou pelo menos não aumente), principalmente em arquivos que apresentem "redundâncias localizadas", isto é, redundâncias distintas em trechos distintos.

A figura III-6 mostra, para vários arquivos com características distintas, a quantidade de informação que já havia sido comprimida no momento em que a TS encheu. Pode-se constatar que usando-se uma TS com 4K entradas (códigos de 12 bits), como a utilizada para obtenção dos dados da figura III-6, a compressão começaria a não se aproveitar de redundâncias em arquivos maiores que 10K bytes em média.

tipo do arquivo	tamanho em bytes	infomação processada antes de TS encher
texto	267.613	9.879
executável	145.424	10.700
programa fonte	354.740	11.486

figura III-6 : quantidade de informação processada necessária para encher a TS

Para aplicações que manipulam longas sequências de caracteres, o método LZW pode não obter um percentual de compressão razoável e em casos extremos pode até expandir os dados ao invés de comprimir. Operações de "backup" de discos ou transferência de longos arquivos são operações que podem não ser suportadas satisfatoriamente pelo método LZW.

O total preenchimento da TS parece indicar uma falha no dimensionamento da mesma. Este problema poderia ser resolvido, para implementações em software, através de uma alocação dinâmica da TS. Porém, o crescimento ilimitado da TS esbarra em vários problemas.

No caso de arquivos onde se apresentam "redundâncias localizadas", apenas os últimos símbolos acrescidos seriam constantemente consultados, tornando a TS um recurso mal utilizado. Outro problema é que neste caso, o tempo necessário para identificar se um símbolo pertence à TS, pode ser proporcional ao seu tamanho e portanto, aumentará. Finalmente restrições de projeto podem, na prática, impossibilitar o crescimento da TS (falta de memória, por exemplo).

As restrições acima citadas e o desejo de produzir implementações em hardware, impõem um tamanho máximo fixo à TS. Partindo agora do principio que a TS deve ter um tamanho fixo, devemos de alguma forma retirar códigos da TS para dar lugar aos novos códigos gerados.

Uma primeira idéia é substituir os primeiros códigos incluidos por novos códigos gerados, ou seja, implementar uma politica FIFO (First In First Out) de atualização da TS. Esta solução contudo, não se aplica, já que desta forma não podemos garantir que o prefixo de um simbolo que pertence à TS também pertence à TS (o mesmo poderia ter sido removido para dar lugar a um novo código) - condição indispensável para o funcionamento do algoritmo.

Uma solução aceitável é reinicializar a TS quando a mesma encher. Assim, quando um novo simbolo tiver de ser incluido na TS e esta estiver cheia, toda a informação nela armazenada será desprezada e novos simbolos (que talvez representem novos tipos de redundância) serão incluidos na mesma, à partir de então.

A incorporação da reinicialização da TS ao procedimento de compressão da técnica LZW é feita de maneira bastante simples, de forma a não comprometer a facilidade de implementação do algoritmo. Para isso, basta que o comando (vide figuras III-2 e III-4)

```
        Inclua código+caractere em TS

    seja substituido por:

        Se TS não estiver cheia
            Inclua código+caractere em TS
        Senão
            Reinicialize TS
```

A substituição indicada acima, deve ser feita tanto no processo de compressão como no processo de descompressão. No processo de compressão a reinicialização deve ser feita para obter o aumento no percentual de compressão discutido anteriormente. Já no processo de descompressão, a reinicialização se faz necessária para possibilitar uma perfeita sincronização entre compressor e descompressor.

A reinicialização da TS pode ser feita de forma total ou parcial. Na reinicialização total apenas os códigos associados aos caracteres do código fonte são mantidos (primeiros 256 códigos). Desta forma, a TS apresenta, após a reinicialização, a mesma configuração que tinha no inicio do processo.

A reinicialização total da TS acarreta uma queda abrupta na capacidade de compressão do algoritmo. Contudo, a recuperação da capacidade de compressão, como acontece também no inicio do processo de compressão, se dá de forma consideravelmente rápida, à medida que mais informação é processada.

Na reinicialização parcial da TS apenas uma fração da TS é desprezada. A idéia da reinicialização parcial é evitar uma queda brusca na capacidade de compressão do algoritmo.

A parte da TS não descartada na reinicialização parcial

entretanto, pode estar associada a um tipo de redundância que não está mais presente nos dados sendo atualmente comprimidos. Dessa forma, a reinicialização parcial não evitará, de maneira apreciável a queda na capacidade de compressão do algoritmo, e por outro lado, contribui para um mau aproveitamento da TS. O mau aproveitamento da TS implica em um aumento na frequência de reinicializações, que por sua vez pode contribuir para uma queda na velocidade de compressão/descompressão e no percentual de compressão atingidos.

Além de comparações do desempenho do algoritmo aqui proposto com os métodos LZW e de Huffman, o capitulo V apresenta comparações do desempenho do algoritmo LZW-MODIFICADO utilizando reinicialização total da TS contra diversas variações na implementação do algoritmo com reinicialização parcial da TS.

Os resultados obtidos indicam que a utilização de reinicialização parcial é ligeiramente melhor que a reinicialização total, para valores de reinicialização menores que a metade da TS, ou seja, quando mais da metade da TS é descartada. Para valores de reinicialização maiores que a metade da TS, a reinicialização total atinge percentuais de compressão maiores que os obtidos pela reinicialização parcial. A reinicialização parcial exige o acréscimo de uma certa complexidade no algoritmo, que no nosso entender, não é compensador. A implementação discutida no capitulo IV utiliza a reinicialização total da TS.

CAPITULO IV

Implementação do algoritmo LZW-MODIFICADO

O algoritmo especificado no capitulo anterior foi implementado na forma de uma biblioteca de funções para compressão de dados. A organização das funções em uma biblioteca permite a fácil utilização do algoritmo LZW-MODIFICADO por utilitários que necessitem de mecanismos de compressão e descompressão de dados, para realização de suas tarefas.

A biblioteca, chamada infocompress*, é detalhada na seção 4.1. A biblioteca infocompress foi utilizada em um compressor de dados, chamado compress**, e em um pacote comercial para comunicação entre ambientes UNIX e MS-DOS, chamado AGIX***. A avaliação do desempenho das funções da biblioteca infocompress em transmissão de dados usando o AGIX é discutida na seção 5.3 do próximo capitulo.

4.1 Biblioteca de funções de compressão LZW-MODIFICADO

Infocompress é uma biblioteca de funções, usada para efetuar a compressão e descompressão de informação, sem que seja necessário um conhecimento a priori do tipo da informação que será processada. As funções da biblioteca utilizam o algoritmo de compressão de dados LZW-MODIFICADO. A biblioteca foi implementada utilizando a linguagem de programação C.

* - infocompress é marca depositada da Infocon Software
** - compress é marca depositada da Infocon Software
*** - AGIX é um produto da Sciencia Informática e Tecnologia e da Infocon Software

As funções da biblioteca operam sobre fluxos de informação, como mostra a figura IV-1.

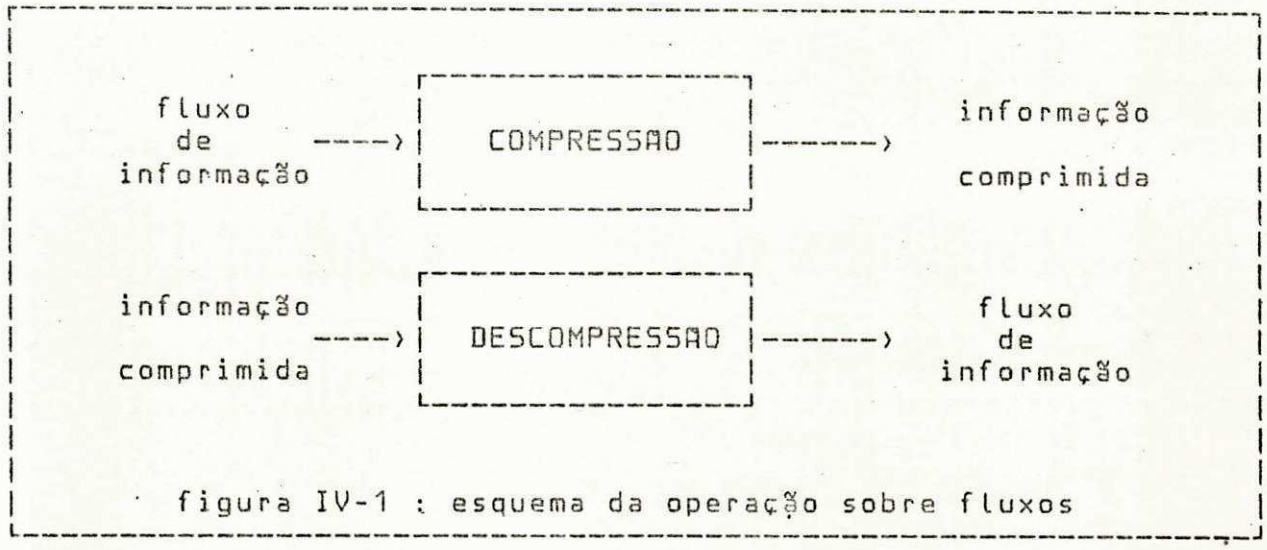


figura IV-1 : esquema da operação sobre fluxos

Um fluxo de informação é uma entidade associada a uma massa de dados processada por uma aplicação, que controla a operação de compressão ou descompressão desta informação. Um fluxo de informação define o tipo de operação que deve ser realizada sobre a informação (compressão ou descompressão), o tamanho da TS utilizada nesta operação, o número de bits dos caracteres na entrada e na saida do fluxo e uma função de saida, responsável pelo escoamento dos códigos resultantes da compressão ou dos caracteres resultantes da descompressão.

A maioria das aplicações opera sobre um único fluxo de informação de cada vez. Entretanto, as funções da biblioteca permitem que uma aplicação opere concorrentemente sobre mais de um fluxo de informação. Um exemplo tipico desta situação, é uma aplicação que comprime a informação de um fluxo, enquanto descomprime a de um outro. Um exemplo com duas tais aplicações é

mostrado na figura IV-2. Observe que os fluxos 1 e 4 são de compressão, enquanto que os fluxos 2 e 3 são de descompressão.

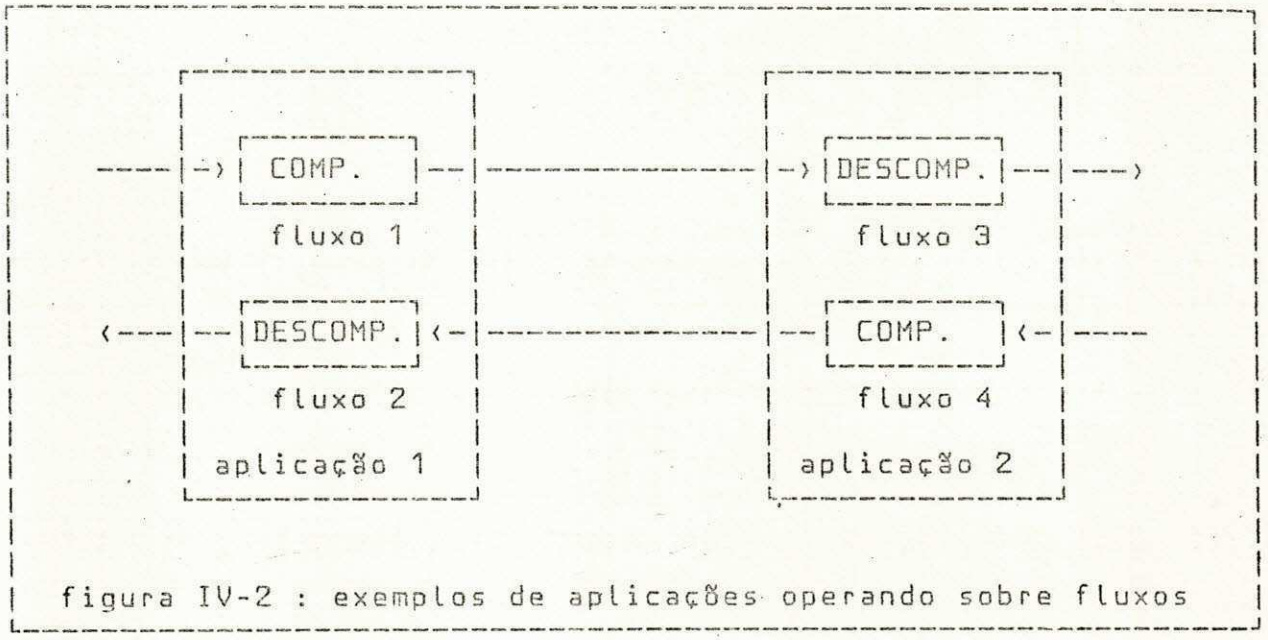


figura IV-2 : exemplos de aplicações operando sobre fluxos

A biblioteca infocompress é composta por seis funções. São elas:

abra_fluxo que possibilita a criação e inicialização de um fluxo de compressão ou descompressão;

comprima que efetua a compressão de um caractere associado a um fluxo de compressão;

descomprima que efetua a descompressão de um caractere associado a um fluxo de descompressão;

stdsaida que é uma função de saída padrão;

fcomprima que recebe como parâmetros, apontadores para arquivos fonte e destino e comprime o arquivo fonte armazenando o resultado da compressão no arquivo destino;

fdescomprima que recebe como parâmetros, apontadores para arquivos fonte e destino e descomprime o arquivo fonte armazenando o resultado da descompressão no arquivo destino;

Para que uma aplicação possa utilizar as funções de compressão ou descompressão da biblioteca, é necessário:

1. definir os parâmetros para criação do fluxo de compressão ou descompressão (número de bits nos caracteres na saída dos fluxos de compressão ou na entrada dos fluxos de descompressão, tamanho da TS, função de saída).

2. definir a função de saída para o fluxo. Este passo pode não ser necessário, se a função stdsaida, definida na biblioteca, satisfizer as necessidades da aplicação.

4.1.1 Compressão de um fluxo de informação

A informação processada em um fluxo de compressão é formada por caracteres de 8 bits. Esses caracteres são comprimidos um a um, usando a função comprima, discutida em detalhes mais adiante. Sequências de caracteres processadas por comprima na entrada são convertidas em códigos com tamanho fixo (função do tamanho da TS), definido na criação do fluxo através da função abra_fluxo.

Os códigos gerados na compressão são escoados por uma função de saída definida pelo usuário, ou pela função de saída padrão (stdsaida) definida na biblioteca. As regras para implementação da função de saída serão discutidas mais adiante.

Como os códigos gerados na compressão são maiores que os caracteres escoados pela função de saida, é necessário "quebrar" um código comprimido em vários caracteres. Os códigos comprimidos podem ser divididos em caracteres de 8 ou 7 bits. A primeira opção (8 bits) seria usada, tipicamente, para armazenar o resultado da compressão em um arquivo em disco, enquanto que a segunda opção (7 bits) seria adequada para a transmissão do resultado através de uma porta de comunicação, onde o oitavo bit seria usado como bit de paridade, e fornecido pelo hardware de comunicação. O número de bits no qual devem ser divididos os códigos comprimidos é também definido na criação do fluxo.

A figura IV-3 resume a operação de compressão de um fluxo:

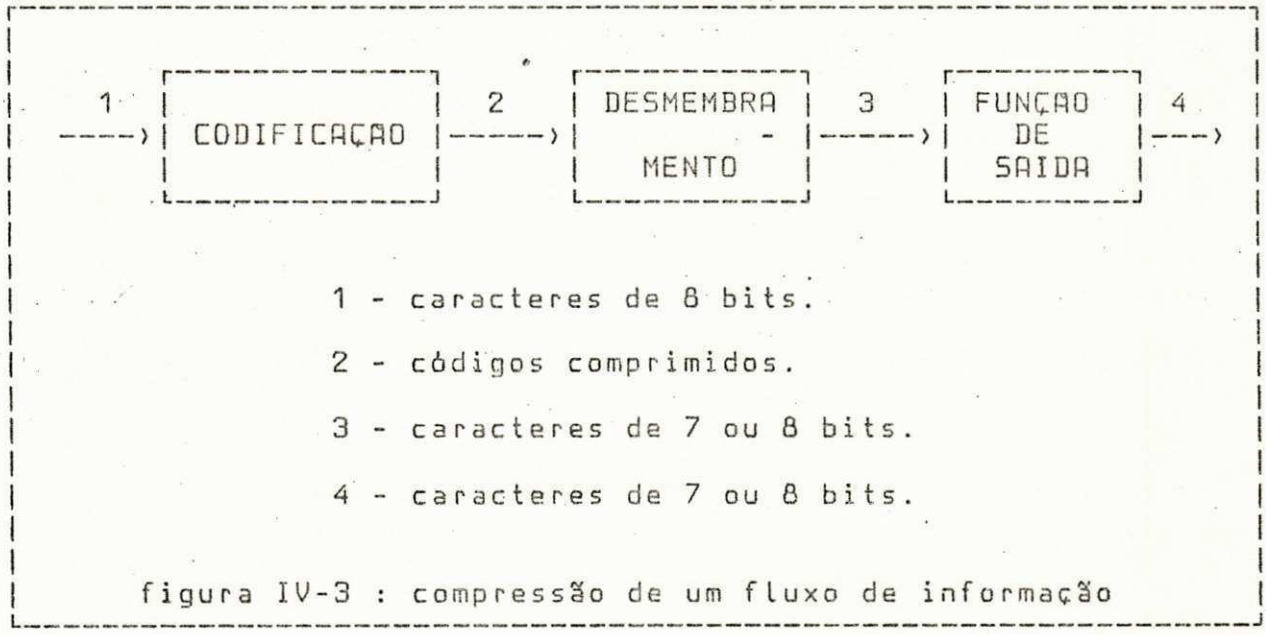


figura IV-3 : compressão de um fluxo de informação

As caixas CODIFICAÇAO e DESMEMBRAMENTO da figura IV-3, correspondem à função comprima.

4.1.2 Descompressão de um fluxo de informação

A informação processada em um fluxo de descompressão é formada por caracteres de 8 ou 7 bits. Os caracteres do fluxo são descomprimidos um a um, pela função descomprima, que também será detalhada mais adiante.

Os caracteres recebidos são concatenados a fim de formarem os códigos gerados no processo de compressão. O número de bits dos caracteres que entram no fluxo, bem como o tamanho dos códigos gerados através da concatenação de vários caracteres, são definidos na criação do fluxo de descompressão (função abra_fluxo), e devem "casar" com os valores usados no processo de compressão correspondente.

Os códigos são então decodificados, gerando uma sequência de caracteres de 8 bits, que representa a informação original. Tal informação é entregue, caractere a caractere, para uma função de saida, que apresenta as mesmas caracteristicas da função de saida de um fluxo de compressão, e que é responsável pelo processamento final da informação.

A figura IV-4 resume a operação de descompressão de um fluxo:

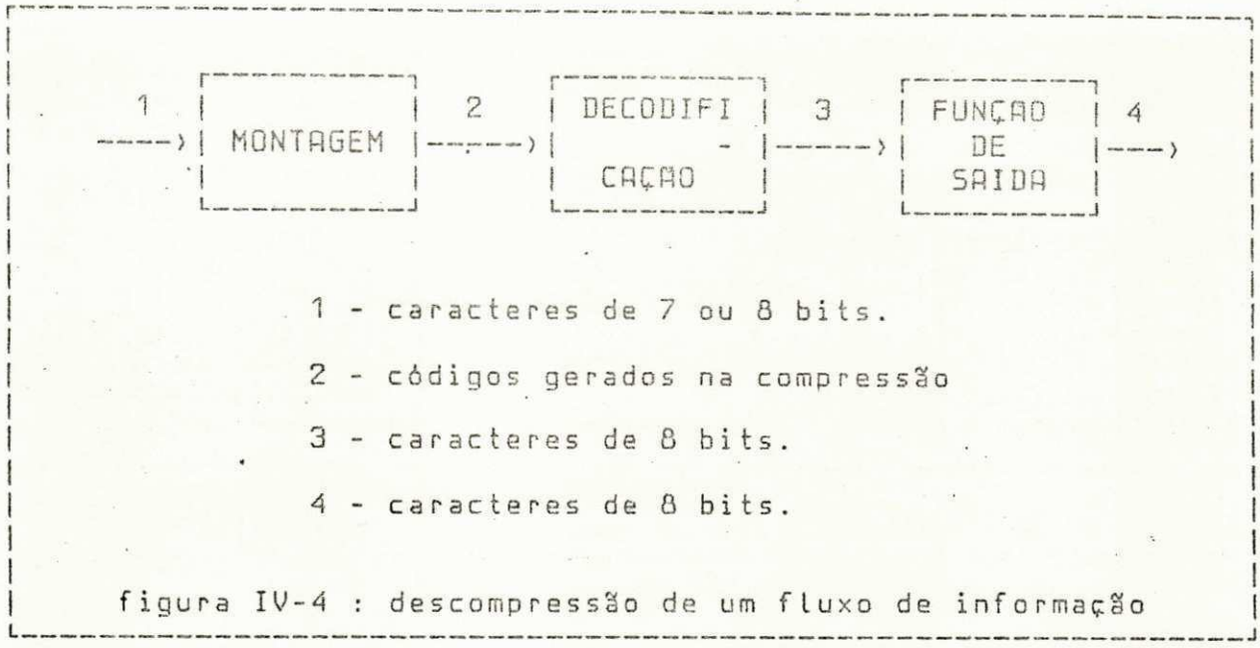


figura IV-4 : descompressão de um fluxo de informação

As caixas MONTAGEM e DECODIFICAÇAO da figura IV-4, correspondem à função descomprima.

4.1.3 Descrição da biblioteca

As funções da biblioteca serão discutidas em detalhes a seguir.

a) Função para criação e inicialização de um fluxo

Antes de iniciar a compressão ou descompressão da informação, é necessário criar um fluxo correspondente. A função abra_fluxo é responsável por esta operação.

Um fluxo é caracterizado por 4 atributos :

1. o tipo de fluxo (compressão ou descompressão).

2. o tamanho dos códigos gerados na compressão, denotado por tam_cod (tam_cod = log N, onde N é o tamanho da TS).

3. o número de bits significativos nos caracteres de saída de um fluxo de compressão ou nos caracteres de entrada de um fluxo de descompressão, denotado por nbits (pode ser 7 ou 8 bits).

4. a função de saída, responsável pelo tratamento final da informação comprimida ou descomprimida.

Sintaxe

```
C_ID_FLUXO abra_fluxo( tipo, tam_cod, nbits, fun_sai, param)
char  tipo;
short tam_cod;
char  nbits;
int   (*fun_sai)();
long  param;
```

Semântica

A função abra_fluxo cria um novo fluxo de compressão (se tipo for igual C_COMP) ou de descompressão (se tipo for igual a C_DESCOMP).

A informação comprimida será representada por códigos de tamanho tam_cod bits (9 <= tam_cod <= 16).

Na compressão, os códigos de tam_cod bits gerados, serão desmembrados em caracteres de nbits bits, que serão escoados pela função de saída apontada por fun_sai. A descompressão por sua vez, restaura os códigos de tam_cod bits gerados na compressão,

através da concatenação de vários caracteres de nbits bits recebidos na entrada. Os códigos gerados são descomprimidos, gerando os caracteres de 8 bits da informação original, que serão escoados por fun_sai.

Tanto na compressão quanto na descompressão, os caracteres serão escoados pela função de saida, com uma chamada:

```
(*fun_sai)( c, param );
```

onde c é o caractere sendo escoado e param é um parâmetro escolhido pelo usuário, que permite que o usuário escreva funções de saida mais gerais. Observe que param não é examinado pelas funções da biblioteca; é apenas passado como argumento para fun_sai, conforme mostramos acima. Sua semântica é escolhida pelo usuário.

Se a função abra_fluxo conseguir criar o fluxo com sucesso, retornará uma identificação de fluxo (do tipo C_ID_FLUXO), se não o valor C_ID_NULO será retornado. Esta identificação é usada nas funções comprima e descomprima para identificar o fluxo desejado.

b) Função para compressão de informação

A compressão de informação é efetuada caractere a caractere, com a função comprima. Esta função é quem realmente implementa o algoritmo de compressão LZW-MODIFICADO. Esta função só pode ser usada com um fluxo do tipo C_COMP.

Sintaxe

```
int  comprima( id, c )
C_ID_FLUXO     id;
int            c;
```

Semântica

O caractere de 8 bits c é enviado para a entrada do fluxo com identificação id para compressão. Os códigos gerados na compressão serão desmembrados em caracteres com nbits bits, que serão escoados pela função de saida. Vale lembrar que nem toda chamada a comprima desencadeará uma chamada à função de saida, pois um código de tam_cod bits corresponde a compressão de uma sequência de um ou mais caracteres. Observe também, que uma chamada a comprima pode desencadear mais de uma chamada à função de saida, já que cada código de tam_cod bits é desmembrado em mais de um caractere de nbits bits.

Se o caractere c a ser comprimido for igual a EOF, o fluxo é fechado e, após o escoamento dos caracteres pendentes, a seguinte função é chamada :

```
(*fun_sai)( EOF, param );
```

EOF tem mais do que nbits significativos, para poder diferenciar este caractere de controle dos caracteres contendo informação, escoados pela função de saida.

Uma vez fechado, o fluxo id não poderá ser usado em chamadas à função comprima.

A função comprima retorna C_ERRO em caso de erro interno ou se a função de saida retornar C_ERRO. Caso contrário, o valor C_OK é retornado.

c) Função para descompressão de informação

A descompressão de informação é efetuada caractere a caractere, pela função descomprima. Esta função implementa o algoritmo de descompressão LZW-MODIFICADO e só pode ser usada com um fluxo do tipo C_DESCOMP.

Sintaxe

```
int  descomprima( id, c )
C_ID_FLUXO      id;
int             c;
```

Semântica

O caractere c é enviado para a entrada do fluxo id para descompressão. Os códigos gerados na compressão são regenerados através da concatenação de vários caracteres de nbits bits. A informação original é gerada pela descompressão dos códigos e é escoada através da função de saída.

Nem toda chamada a descomprima desencadeará uma chamada à função de saída, pois um código de tam_cod bits é formado por mais de um caractere de nbits bits. É também possivel que uma chamada a descomprima venha a desencadear mais de uma chamada à função de saída, já que cada código decodificado por descomprima corresponde a uma sequência de um ou mais caracteres.

O fluxo é fechado com a chamada:

```
descomprima( id, EOF );
```

Como resultado desta chamada, a função de saida será chamada da seguinte forma :

```
(*fun_sai)( EOF, param );
```

para avisar à funçao de saida que nenhuma outra informação será escoada para este fluxo.

A função descomprima retorna C_ERRO em caso de erro interno ou se a função de saida retornar C_ERRO. Caso contrário, o valor C_OK é retornado.

d) Função para escoamento dos caracteres (função de saida)

A função de saida deve ser fornecida pelo usuário, tanto na compressão quanto na descompressão. Esta função deve obedecer às seguintes regras :

1. o primeiro parâmetro recebido é um int e representa o caractere a escoar.

2. o segundo parâmetro pode ser de qualquer tipo, desde que não seja maior que um long. A função usa este parâmetro como desejar (pode até ignorá-lo). Na abertura do fluxo, este parâmetro é informado, assim como a função de saida a ser utilizada.

3. se o primeiro parâmetro tiver valor EOF, nenhum outro caractere será recebido para o fluxo correspondente.

4. se o primeiro parâmetro não for EOF, apenas os primeiros nbits bits do caractere devem ser considerados, se o fluxo é de compressão. No caso de um fluxo de descompressão, os 8 primeiros bits devem ser considerados.

5. o valor C_ERRO deve ser retornado em caso de erro; caso contrário, o valor C_OK deve ser retornado.

Como exemplo de uma função de saída, vamos escrever uma função que armazenará em um arquivo os caracteres recebidos. O apontador do arquivo é recebido como o valor de **param**.

```
int  stdsaida( c, fp )
int  c;       /* caractere a ser escoado */
FILE *fp;     /* apontador do arquivo de saida */
{

     if ( c == EOF ) {
          return(fflush( fp ) == EOF ? C_ERRO : C_OK);
     } else {
          /* o caractere c ja vem mascarado com nbits */
          return(fputc( c, fp) == EOF ? C_ERRO : C_OK);
     }

}
```

A função **stdsaida** faz parte da biblioteça **infocompress**, e pode ser usada como parâmetro para a função **abra_fluxo** se desejado.

e) Funções para compressão e descompressão de arquivos

A compressão de arquivos é uma operação bastante frequente nos utilitários que potencialmente usarão as funções da biblioteca **infocompress**. Para facilitar estas operações, duas funções

foram acrescidas; são elas fcomprima e fdescomprima.

A função fcomprima comprime o arquivo aberto apontado por fpe e coloca o resultado no arquivo aberto apontado por fps.

Já a função fdescomprima descomprime o arquivo aberto apontado por fpe e coloca o resultado no arquivo aberto apontado por fps.

Sintaxe

```
int  fcomprima( fpe, fps, nbits )
FILE            fpe;
FILE            fps;
char            nbits;
```

Semântica

E aberto um fluxo de compressão utilizando nbits bits nos caracteres escoados na saida e a função stdsaida como função de saida; fps é o parâmetro da função de saida; o valor 12 é usado para o parâmetro tam_cod. Em seguida o arquivo apontado por fpe é lido até o final, e cada caractere lido é comprimido através da função comprima. Se algum erro for detectado no decorrer do processo, a função retorna C_ERRO, caso contrário retorna C_OK.

A função fdescomprima tem a mesma sintaxe e semântica da função fcomprima, obviamente substituindo o fluxo de compressão e a função de compressão por um fluxo de descompressão e a função de descompressão.

4.1.4 Utilização da biblioteca

Uma vez especificada e implementada a biblioteca de compressão de dados, resta especificar e implementar os utilitários que usarão as funções da biblioteca. O exemplo a seguir apresenta a utilização da função fcomprima no módulo de compressão de um utilitário de compressão de arquivos bastante simples.

Sintaxe : comp [arqent [arqsai]]

1. comp

   comprime a entrada padrão, colocando o resultado na saída padrão.

2. comp arqent

   comprime o arquivo arqent, colocando o resultado na saída padrão.

3. comp arqent arqsai

   comprime o arquivo arqent, colocando o resultado em arqsai.

```
#include  <stdio.h>
#include  <infocmpr.h>

main( argc, argv )
int  argc;
char *argv[];
{
     FILE *fpe, *fps;

     if( argc > 3 ) {
          printf("Uso : comp [arqent [arqsai]]\n");
          exit( 1 );
     }
     if( argc >= 2 ) {
          if( ( fpe = fopen( argv[1], "r" ) ) == NULL ) {
               printf("comp: nao pode abrir %s\n",argv[1]);
               exit( 1 );
          }
     } else {
          fpe = stdin;
     }
     if( argc == 3 ) {
          if( ( fps = fopen( argv[2], "w" ) ) == NULL ) {
               printf("comp: nao pode criar %s\n",argv[2]);
               exit( 1 );
          }
     } else {
          fps = stdout;
     }
     if( fcomprima( fpe, fps, 8 ) == C_ERRO ) {
          printf("comp: erro na compressao\n");
          exit( 1 );
     }
     exit( 0 );
}
```

Alguns simbolos especiais são definidos no arquivo **infocmp.h**, que deve ser incluido nos programas que utilizem as funções da biblioteca **infocompress**. Estes simbolos são:

C_ID_FLUXO — tipo de um identificador de fluxo, retornado pela função abra_fluxo.

C_ID_NULO — valor retornado pela função abra_fluxo, quando a mesma não consegue criar um fluxo.

C_TAM_COD	valor sugerido para tam_cod (12 bits).
C_COMP	identifica um fluxo de compressão.
C_DESCOMP	identifica um fluxo de descompressão.
C_ERRO	valor retornado pela maioria das funções, em caso de erro.
C_OK	valor retornado pela maioria das funções, em caso de sucesso.

4.2 Detalhes da implementação

A biblioteca infocompress e o utilitário de compressão compress foram desenvolvidos totalmente em linguagem C, em ambiente UNIX, sendo posteriormente transportados para o ambiente MS-DOS. A figura IV-5 mostra o tamanho de alguns dos módulos desenvolvidos.

projeto	arquivo	tamanho (bytes)
infocompress (versão 1)	ab_fluxo.c	4915
infocompress (versão 1)	comprima.c	2342
infocompress (versão 1)	descprma.c	3814
infocompress (versão 1)	fcomprma.c	1467
infocompress (versão 1)	fdescpma.c	1499
infocompress (versão 1)	stdsaida.c	849
infocompress (versão 1)	todos os arq.	34738
infocompress (versão 2)	fcomprma.c	7810
infocompress (versão 2)	fdescpma.c	7443
infocompress (versão 2)	todos os arq.	20002
compress	todos os arq.	111041

figura IV-5 : tamanho dos módulos desenvolvidos

Das funções e procedimentos implementados para construção da biblioteca e dos utilitários, sem dúvida as mais importantes são as funções comprima e descomprima, responsáveis pelas operações de compressão e descompressão do algoritmo LZW-MODIFICADO. A implementação destas funções será analisada em detalhes nesta seção.

Como apresentado anteriormente, as operações de compressão e descompressão são realizadas sobre fluxos de informação. Um fluxo é criado através da função abra_fluxo. Internamente um fluxo é representado pela estrutura de dados ilustrada na figura IV-6. Os parâmetros recebidos pela função abra_fluxo inicializam a estrutura de forma conveniente. A função abra_fluxo se tiver êxito na sua execução, retorna um apontador para o fluxo criado. Este apontador será utilizado pelas funções de compressão e descompressão, para referenciar no futuro, o fluxo associado.

A estrutura de dados apresentada na figura IV-6 serve para representar fluxos de compressão e fluxos de descompressão. Os campos da estrutura estão divididos em seis grupos (vide figura IV-6) e serão detalhados a seguir.

```
struct s_fluxo {

/* GRUPO I   */   char     tipo;
                  short    tam_cod;
                  char     nbits;
                  int      (*fun_sai)();
                  long     param;

/* GRUPO II  */   char     n_bits_pend;
                  short    bits_pend;

/* GRUPO III */   short    tam_tab;
                  short    max_tam_tab;
                  char     nao_foi_ref;

/* GRUPO IV  */   short    *pref_ts;
                  char     *extn_ts;

                  union    u_tipo_fluxo {

/* GRUPO V   */                struct  s_compressao {
                                         char    erro;
                                         short   prefixo;
                                         short   *hash;
                                   } compressao;

/* GRUPO VI  */                struct  s_descompressao {
                                         short   cod_velho;
                                         char    ult_c_ext;
                                         short   topo;
                                         short   tam_pilha;
                                         char    *pilha;
                                   } descompressao;

                        } tipo_fluxo;
              };
```

figura IV-6 : estrutura de dados de um fluxo

O primeiro grupo de campos define as caracteristicas do fluxo criado. Estes campos são inicializados com os valores passados como parâmetros para a função abra_fluxo. O campo tipo identifica o tipo do fluxo, ou seja, se é um fluxo de compressão (cujo valor é C_COMP) ou um fluxo de descompressão (culo valor é C_DESCOMP). O campo tam_cod define o tamanho dos códigos e

consequentemente o tamanho da TS que será usada. O campo nbits indica o número de bits que realmente contém informação em cada caractere enviado pela função de saida, se o fluxo é de compressão, ou nos caracteres recebidos pela função na entrada, se o fluxo é de descompressão. O campo fun_sai é um apontador para a função de saida, responsável pelo escoamento dos caracteres comprimidos ou descomprimidos enquanto que param é um parâmetro para esta função.

Como os códigos emitidos na compressão e recebidos na descompressão são maiores que um caractere, é necessário mecanismos que permitam o armazenamento temporário de bits pendentes em um fluxo de compressão ou descompressão. O grupo de campos seguinte formado pelos campo bits_pend e n_bits_pend provê estes mecanismmos. Os bits pendentes são armazenados no campo bits_pend, enquanto que o campo n_bits_pend armazena o número de bits pendentes.

Em seguida temos alguns campos de controle. nao_foi_ref é usado para indicar se é a primeira vez que aquele fluxo está sendo referenciado, este campo é importante também para o processo de reinicialização. Os campos tam_tab e max_tam_tab indicam respectivamente, até que ponto a TS foi preenchida, e o tamanho máximo da TS.

A TS é representada pelos campos pref_ts que armazena os prefixos e extn_ts que armazena os caracteres de extensão. Na abertura do fluxo a TS é inicalizada. A inicialização consite na atribuição dos valores correspondentes às primeiras 256 posições

da tabela apontada por extn_ts.

Os dois últimos grupos são mutuamente exclusivos, ou seja, dependendo do tipo de fluxo é utilizado um ou outro grupo de campos. Se o fluxo é de compressão os campos utilizados são: erro utilizado para prover mecanismos de recuperação de erro, prefixo utilizado para armazenar o prefixo que representa a sequência de caracteres processada a cada passo da compressão e hash utilizado para implementar mecanismos de acesso direto à TS.

Se o fluxo é de descompressão, os campos são: cod_velho que armazena o último código processado, ult_c_ext que armazena o último caractere de extensão processado, topo, pilha e tam_pilha utilizados para implementação de uma pilha, necessária para inversão da sequência de caracteres geradas no processo de descompressão.

4.3 Otimizações do perfil de execução da implementação proposta

A primeira implementação dos algoritmos de compressão e descompressão LZW-MODIFICADO baseou-se na estrutura de dados discutida na seção anterior.

Embora os percentuais de compressão atingidos fossem plenamente satisfatórios e os testes com vários tamanhos e tipos de arquivos indicassem um ganho bastante considerável no percental de compressão do algoritmo LZW-MODIFICADO frente ao algoritmo LZW, e mesmo sendo a velocidade de 4100 caracteres por segundo, alcançada pelo algoritmo LZW-MODIFICADO, compatível com velocidades de compressão do algoritmo LZW apresentadas por outras

literaturas, como [BELL 86]*, algumas aplicações necessitam de uma velocidade de compressão maior do que as atingidas.

O problema de velocidade apresentado pela implementação do algoritmo LZW-MODIFICADO não está vinculado às modificações incluidas no algoritmo LZW. A avaliação de desempenho apresentada na seção 5.2 do capitulo V, garante esta afirmação. Algumas caracteristicas do próprio algoritmo LZW devem ser cuidadosamente observadas, para que uma implementação deste algoritmo, tenha uma velocidade de compressão/descompressão que satisfaça as exigências de determinadas aplicações.

Para cada caractere a ser comprimido é necessário um acesso à TS, portanto esta é uma operação crítica, e merece uma atenção especial.

Vários perfis de execução foram feitos sobre a biblioteca resultante da primeira implementação. Estes perfis acusaram o consumo de uma elevada fração do tempo total de execução, pelas rotinas que realizam o acesso à TS. Cerca de 45% do tempo total de execução era gasto nestas rotinas.

* - Bell em [BELL 86] alcançou velocidades de 5700 caracteres por segundo em uma implementação em software do algoritmo LZW, executando em um VAX 11/750. A diferença de aproximadamente 30% entre as velocidades das implementações, corresponde perfeitamente à diferença na velocidade de execução das máquinas.

A figura IV-7 representa o perfil da execução de um utilitário de compressão de arquivos que utiliza as funções da primeira implementação da biblioteca infocompress. O número de chamadas à rotina comprima indica o tamanho do arquivo que foi submetido à compressão. A velocidade de compressão atingida neste exemplo foi de 4205 caracteres por segundo. O número de chamadas à rotina _rastreio indica que ocorreram 15420 colisões durante o processo, o que constitui um percentual de 28%. O tempo gasto no acesso à TS é dado pela soma dos tempos das rotinas _pesquisa e _rastreio (5.35 segundos, o que representa aproximadamente 41% do tempo de execução).

nome da rotina	percentual de tempo gasto	tempo acumulado	número de chamadas
_pesquisa	27,6	3,63	55295
comprima	27,2	7,21	55297
_rastreio	13,1	8,93	15420
stdsaida	10,6	10,32	35119
fcomprima	10,5	11,70	1
_desmembra	9,9	13,00	23413
abra_fluxo	1,1	13,15	1
main	0,0	13,15	1

figura IV-7 : perfil da execução de um utilitário de compressão

O perfil mostra que um caminho bastante promissor para diminuir o tempo de execução e consequentemente aumentar a velocidade de compressão, é a otimização da parte do código relativa ao acesso à TS. As operações de desmembramento de códigos (rotina _desmembra) e abertura de fluxo (rotina abra_fluxo), responsáveis por cerca de 11% do tempo total de execução, também podem ser alvos de otimizações.

Diversos caminhos foram tentados para se obter um ganho no tempo total de execução. A maioria das soluções vislumbradas traziam consigo algumas desvantagens. A implementação de tais soluções dependeu portanto de uma análise de custo X benefício de cada solução.

A codificação em assembler de rotinas críticas como as funções de hash, embora pudesse potencialmente aumentar bastante a velocidade de execução destas funções, e do algoritmo como um todo, têm o grave problema de diminuir a portabilidade da biblioteca e foi portanto descartada.

O tamanho do código indicado na abertura de um fluxo (tam_cod) é um parâmetro que interfere no percentual de compressão associado àquele fluxo. No inicio do processo de compressão, quando a informação contida na TS é somente os 256 caracteres inicialmente armazenados, ocorre uma expansão da informação, já que caracteres de 8 bits serão substituidos por códigos de tam_cod bits. Contudo, quando uma certa quantidade de informação é acrescida à TS, sequências de mais de um caractere serão substituidas por códigos de tam_cod bits, de forma a comprimir a informação. A utilização de um código muito pequeno faz com que a expansão inicial seja minima, entretanto o pequeno tamanho da TS é responsável por uma frequência muito grande de reinicializações, comprometendo o percentual de compressão. A utilização de um código muito grande produz uma expansão acentuada no inicio do processo, que pode impossibilitar a compressão de arquivos pequenos. A figura IV-8 faz uma análise do percentual de

compressão de um arquivo contendo texto, com 10.000 bytes, frente ao tamanho da TS usada.

tamanho do código	nbits usado	tempo de comp.	tempo de descomp.	percentual de compressão
9	7	3.33	2.65	30.16
9	8	3.31	2.55	38.89
10	7	3.12	2.55	39.05
10	8	2.97	2.43	46.67
11	7	3.04	2.38	44.93
11	8	3.52	**2.27**	51.82
12	7	3.12	2.45	46.72
12	**8**	**3.02**	2.53	**53.38**
13	7	3.06	2.54	42.28
13	8	3.14	2.43	49.49
14	7	3.52	2.67	37.84
14	8	3.42	2.66	45.61

figura IV-8 : percentual de compressão X tamanho da TS

A análise apresentada na figura IV-8 aponta o melhor percentual de compressão médio para os fluxos que utilizam **tam_cod** igual a 12 bits.

Uma otimização nos tempos das rotinas de criação de fluxo e desmembramento de códigos gerados baseia-se na fixação a priori do valor utilizado para tam_cod. A utilização de tam_cod fixo e igual a 12 permite substituir a alocação dinâmica da TS e da tabela de hash por uma alocação estática. O tamanho máximo da TS é agora um valor fixo e conhecido podendo ser removido da estrutura de dados associada ao fluxo. O ganho mais substancial entretanto, é obtido na simplificação da rotina que desmembra um código de tam_cod bits em caracteres de nbits bits. Como nbits geralmente é 8 a utilização de tam_cod igual a 12 (8 + 4)

simplifica bastante o código desta rotina, responsável por aproximadamente 10% do tempo total de execução (veja o tempo da rotina _desmembra na figura IV-7).

A dificuldade da obtenção de uma boa função de hash é responsável pela alta taxa de colisões apresentada pela primeira implementação da biblioteca. Várias funções de hash foram implementadas e testadas sem todavia, apresentar resultados satisfatórios. Uma solução que surtiu efeito, embora em troca de uma pequena queda no percentual de compressão, foi o descarte dos símbolos que colidiam. Segundo o algoritmo original, toda sequência de caracteres encontrada na informação sendo comprimida, que não pertencesse à TS, deveria ser incluida na mesma. Entretanto a não inclusão de algumas destas sequências não interfere no funcionamento do algoritmo, desde que na descompressão estas sequências também não sejam incluidas na TS. Dependendo da frequência com que estas sequências são descartadas, a queda no percentual de compressão do algoritmo não é considerável.

O descarte dos símbolos que colidem elimina a necessidade da rotina _rastreia, concorrendo para um aumento na velocidade de compressão. Para que as sequências de caracteres descartadas na compressão também sejam descartadas na descompressão é necessário introduzir no processo de descompressão a mesma função de hash utilizada na compressão. A introdução da função de hash na descompressão implica em um aumento muito pequeno na complexidade do algoritmo e na velocidade de descompressão, sendo portanto plenamente justificável. A figura IV-9 faz comparações da veloci-

dade e percentual de compressão obtidos por implementações com e sem tratamento de colisões. Os valores apresentados justificam o uso desta solução para aumentar a velocidade de compressão.

com tratamento de colisões		sem tratamento de colisões	
velocidade	percentual	velocidade	percentual
7606 c/s	41,86%	9204 c/s	30,30%
8363 c/s	49,41%	9696 c/s	41,78%
8308 c/s	45,13%	10010 c/s	41,29
6450 c/s	51,16%	7952 c/s	46,86
5682 c/s	53,37%	6667 c/s	46,79

figura IV-9 : velocidade e percentual de compressão com e sem tratamento de colisões

A modularização do projeto, caracteristica de uma programação estruturada, permite clareza e facilidade de modificação do código gerado, ao passo que concorre para a proliferação de pequenas funções com objetivos especificos que são chamadas por outras com um objetivo mais geral. Esta interação entre as funções é responsável por uma grande quantidade de chamada de funções e passagem de parâmetros. A experiência com outras implementações utilizando linguagem C em ambiente UNIX mostra que o "overhead" na passagem de parâmetros é muito grande. Este problema se agrava quando funções pequenas, que tem seu corpo executado rapidamente, são chamadas muitas vezes durante a execução de um programa. Neste caso o tempo gasto em processamento útil (execução dos comandos da rotina propriamente) é muito pequeno em relação ao "overhead" de passagem de parâmetros.

Algumas rotinas da biblioteca infocompress apresentam as características indesejáveis mencionadas acima. As rotinas comprima e _pesquisa são chamadas para cada caractere a ser comprimido, a rotina _desmembra é chamada para cada código gerado e a rotina responsável pelo escoamento dos caracteres (função de saida) é chamada uma ou mais vezes para cada código gerado. Este excessivo número de chamadas poderia ser reduzido se as rotinas fossem todas embutidas em uma única rotina. Esta solução tem o grande incoveniente de perder toda a flexibilidade fornecida pela interface da biblioteca, além de não constituir uma boa prática de programação. Entretanto, os ganhos obtidos (a velocidade mais que dobrou) compensaram sua implementação.

A nova implementação da biblioteca, incorporando as otimizações discutidas, consiste de apenas duas funções, uma para comprimir (fcomprima) e o outra para descomprimir (fdescomprima).

Para reduzir os prejuizos ocasionados na interface da biblioteca, as funções fcomprima e fdescomprima tiveram suas sintaxe e semântica um pouco modificadas. A sintaxe e a semântica destas funções são discutidas abaixo.

Sintaxe

```
int  fcomprima( fun_ent, prm_ent, fun_sai, prm_sai, nbits )
int             (*fun_ent)();
long            prm_ent;
int             (*fun_sai)();
long            prm_sai;
char            nbits;
```

Semântica

fun_ent é um apontador para uma função de entrada. Esta função retorna um caractere para a função fcomprima que irá comprimi-lo. A função fcomprima chama a função de entrada até que esta lhe retorne EOF. Neste ponto, a operação de compressão termina. prm_ent é um parâmetro que é passado para a função de entrada. fun_sai, prm_sai e nbits, têm a mesma funcionalidade especificada para os parâmetros de mesmo nome definidos na função abra_fluxo da primeira biblioteca. Se fun_ent ou fun_sai tiver o valor NULL, serão usadas funções definidas internamente para executar a leitura e gravação dos caracteres respectivamente. Se algum erro ocorrer durante o processo de compressão a função fcomprima retorna C_ERRO, caso contrário retorna C_OK. A função fdescomprima tem sintaxe e semântica análogas à de fcomprima.

A avaliação de desempenho feita no capitulo seguinte utilizou a segunda versão da biblioteca infocompress discutida nesta última seção. Os ganhos na velocidade de compressão do algoritmo LZW-MODIFICADO podem ser conferidos nos resultados apresentados no próximo capitulo.

CAPITULO V

Avaliação de desempenho

5.1 Definição do ambiente de avaliação

O primeiro passo para fazer uma avaliação do desempenho do algoritmo LZW-MODIFICADO, foi determinar que métodos deveriam ser confrontados com o algoritmo. O método LZW obviamente seria um dos métodos utilizados no confronto. O outro método escolhido para comparação foi o método de Huffman. O uso do método de Huffman em "benchmarks" de algoritmos de compressão é bastante comum, o que estimulou a sua utilização.

A comparação com o método de Huffman permite que possam ser induzidas comparações com outros métodos, já que o método de Huffman é quase um padrão em avaliações de desempenho de métodos de compressão.

As implementações foram executadas em uma máquina ED-680 da EDISA com microprocessador 68000 e memória de 2 Mbytes, sob o sistema operacional EDIX. Esta máquina tem uma capacidade de processamento de aproximadamente 0.7 MIPS.

O utilitário pack do EDIX foi utilizado para obter os resultados do método de Huffman. Este utilitário implementa um algoritmo de compressão em dois passos similar ao apresentado por Huffman em [HUFF 52].

Os arquivos utilizados na ánalise de desempenho foram os seguintes:

Manual - (267.613 bytes) manual da biblioteca INFOSCREEN, contendo texto, figuras e trechos de programas escritos em C.

Apêndice - (11.610 bytes) um dos apêndices do manual descrito acima.

C - (347.949 bytes) concatenação de vários arquivos contendo programas escritos em C.

Iltd - (122.544 bytes) módulo executável do programa ILTD.

Backup - (4 mega bytes) "dump" de um disco, correspondente a diretórios de usuários do sistema onde estava sendo desenvolvido o algoritmo LZW-MODIFICADO e os testes de desempenho.

As medidas de desempenho utilizadas foram:

a) Indice de Redução (%) definido por:

$$IR = 100 \times \frac{(S_o - S_c)}{S_o}$$

onde S_o é o tamanho do arquivo original e S_c é o tamanho do arquivo comprimido.

b) Tempo de Resposta Total (TRT) o qual é a soma do tempo de compressão e tempo de descompressão em segundos.

5.2 Comparação do LZW-MODIFICADO com outras técnicas de compressão

Sob o ambiente definido na seção anterior, foram calculados os índices de redução e tempos de resposta total dos arquivos anteriormente descritos para as implementações dos algoritmos LZW-MODIFICADO, LZW e HUFFMAN. Os resultados obtidos estão sumariados na tabela da figura V-1.

ARQUIVO	HUFFMAN		LZW		LZW-MODIFICADO	
	IR	TRT	IR	TRT	IR	TRT
manual	38.60	115.57	40.37	58.00	49.41	55.60
apêndice	40.70	5.32	51.16	2.75	51.16	2.78
c	32.10	166.89	39.73	74.85	45.77	74.53
iltd	26.10	61.37	26.01	28.29	40.25	26.92
backup	32.80	1977.99	14.47	1014.02	48.72	928.92

IR expresso em percentual
TRT expresso em segundos

figura V-1 : IR e TRT para vários arquivos

Pode-se observar que quanto mais "redundâncias localizadas" apresentarem os arquivos (iltd, backup) melhor será o índice de compressão de LZW-MODIFICADO frente aos demais. Observa-se também que, em arquivos onde a redundância é praticamente constante (manual, c), a diferença do índice de compressão de LZW e LZW-MODIFICADO é pequena, contudo ambos alcançam índices melhores que HUFFMAN. E finalmente, arquivos pequenos (como apêndice, com

aproximadamente 10K) não são suficientes para encher a TS, e portanto os índices alcançados por LZW e LZW-MODIFICADO são os mesmos.

Para os arquivos considerados, o algoritmo LZW-MODIFICADO apresenta um ganho médio geral no Índice de Redução de 39.8% sobre HUFFMAN e 65.8% sobre LZW. Quanto ao Tempo de Resposta Total, os ganhos médios gerais foram 52.8% e 3.3% respectivamente. A figura V-II apresenta um gráfico de barras, onde a relação entre os valores obtidos para o índice de redução podem ser melhor avaliados.

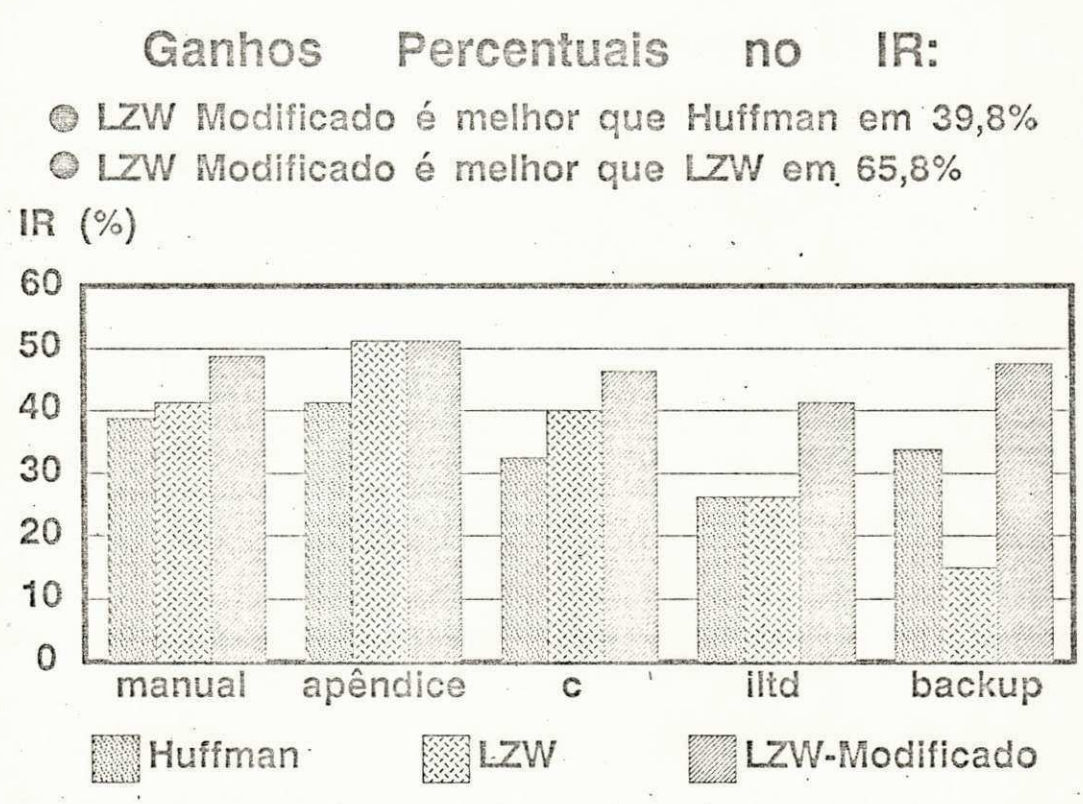


figura V-2 : gráfico de barras - índice de redução

Os gráficos das figuras V-3 e V-4, representam a variação do índice de Redução com o tamanho de arquivos. O gráfico da figura V-3 ilustra o Indice de Redução obtido em um arquivo resultante de sucessivas concatenações dos primeiros 10K bytes do arquivo apêndice. O Indice de Reduçao do arquivo backup foi medido em vários pontos, para obter os valores indicados no gráfico da figura V-4.

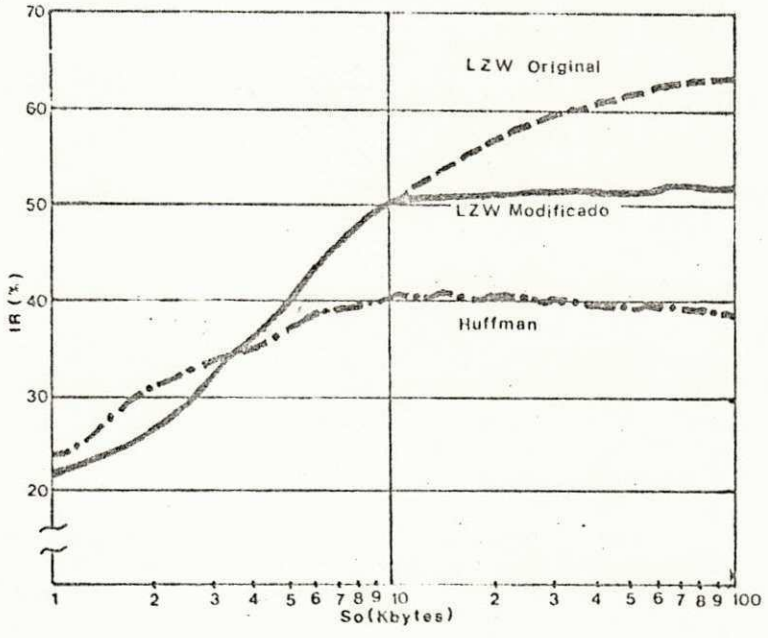


figura V-3 : (IR x S_o) concatenações sucessivas dos primeiros 10K bytes do arquivo apêndice

O arquivo gerado pela concatenação de blocos iguais (figura V-3), favorece a compressão LZW, mas este não é um caso de muito interesse na prática. O Indice de Redução para HUFFMAN, nesse caso, é o mesmo sempre que o tamanho do arquivo é um múltiplo de

10K (desprezando-se o percentual referente à tabela usada na decodificação, que deve ser armazenada junto com as informações comprimidas), já que nesses pontos a distribuição de frequência dos caracteres no arquivo é igual.

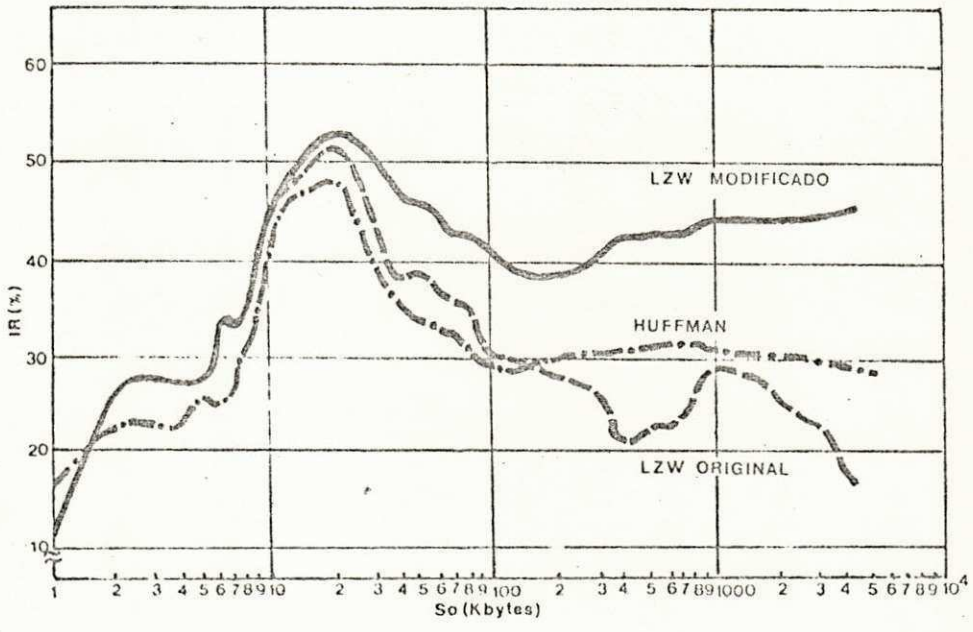


figura V-4 : (IR x S_o) arquivo backup

Arquivos que apresentam tipos de redundâncias distintos em trechos distintos (figura V-4) favorecem a compressão LZW-MODIFICADO, pois a reinicialização da TS permite que um novo tipo de redundância seja utilizado.

5.3 Avaliação do algoritmo LZW-MODIFICADO em transmissão de dados

Sob o mesmo ambiente descrito na seção 5.1, foram feitas transmissões de arquivos, em uma linha de comunicação trabalhando a 9600 bauds, utilizando as versões 1.1 e 2.1 do utilitário AGIX. A tabela da figura V-5, onde os tempos são expressos em segundos e as velocidades em caracteres por segundo, ilustra os resultados obtidos.

ARQUIVO	VERSAO 1.1 (sem comp.)		VERSAO 2.1 (com comp.)	
	TEMPO	VELOCIDADE	TEMPO	VELOCIDADE
manual	400.02	669	166.53	1607
apêndice	16.81	664	7.56	1475
c	520.88	668	157.94	2203
iltd	183.72	667	134.81	909

TEMPO expresso em segundos
VELOCIDADE expressa em bytes por segundo

figura V-5 : desempenho em transmissão de dados

A versão 1.1, que não utiliza compressão de dados, apresenta uma velocidade de transmissão praticamente constante e em média aproximadamente igual a 667 caracteres por segundo. Já a versão 2.1, que utiliza as funções de compressão da biblioteca **infocompress**, apresenta uma velocidade de transmissão variável, que depende do percentual de compressão obtido em cada arquivo.

Os arquivos maiores e com maior taxa de compressão (manual, c) conseguem uma velocidade de transferência maior que os arquivos pequenos e/ou com baixas taxas de compressão (apêndice, iltd). Para os arquivos analisados, a versão 2.1 apresentou uma velocidade de transmissão média efetiva de 1548 caracteres por segundo. Esta velocidade é superior à própria velocidade da linha (1200 caracteres por segundo, desprezando-se possiveis "overheads"). Este aparente absurdo, se deve ao fato do cálculo da velocidade de transferência ser feito sobre o tamanho original dos arquivos e não sobre a quantidade de informação que realmente passou no canal de comunicação.

5.4 Avaliação da reinicialização parcial da TS

A figura V-6 apresenta o percentual de compressão alcançado para vários arquivos, utilizando reinicialização total e reinicialização parcial em 512, 1024, 2048 e 3072 bytes.

ARQUIVO	LZW-M	512	1024	2048	3072
manual	49,41%	50,27%	50,74%	49,99%	45,96%
apêndice	51,16%	51,16%	51,16%	51,16%	51,16%
c	45,13%	45,99%	46,27%	46,22%	40,97%
iltd	41,86%	42,06%	42,67%	41,14%	37,26%

figura V-6 : percentual de compressão com reinicialização parcial

A reinicialização parcial consegue melhores indices de

redução quando mais da metade dos códigos da TS são descartados (512, 1024, 2048). Para os arquivos usados, a reinicialização em 512, 1024 e 2048 apresenta um ganho médio no indice de redução de 1,15%, 1,16% e 0,58% respectivamente. Quando mais da metade dos códigos da TS são mantidos (3072) a reinicialização total é mais vantajosa. Para os arquivos usados, a reinicialização total atinge um indice médio de redução 6,80% melhor que a reinicialização em 3072.

A reinicialização parcial portanto, apresenta apenas em alguns casos, um indice de redução um pouco melhor que a reinicialização total.

Para implementar a reinicialização parcial entretanto, é necessário um cuidado especial na sincronização entre compressor e descompressor. Os mecanismos necessários para prover a sincronização são responsáveis por um incremento na complexidade do algoritmo, que devido ao pequeno ganho obtido no indice de redução, torna a reinicialização parcial desnecessária.

5.5 Avaliação da utilização de run-length encoding associado com LZW-MODIFICADO

A utilização de mais de uma estratégia de compressão em um mesmo algoritmo é comum. Alguns métodos de compressão mais simples, quando utilizados em conjunto com outros métodos mais complexos podem, sem uma perda significativa na velocidade de compressão, aumentar bastante as taxas de compressão obtidas. O método run-lenght encoding dicutido na seção 2.2.2.4 do capitulo II, pela sua simplicidade, constitui um dos métodos mais usados

neste tipo de associação.

Embora os percentuais de compressão alcançados pela implementação do algoritmo LZW-MODIFICADO fossem plenamente satisfatórios, tentou-se aumentar um pouco o percentual de compressão com a introdução de rotinas que implementassem run-length encoding na informação a ser comprimida.

A interface da segunda versão da biblioteca infocompress permitiu a fácil inclusão das rotinas que implementaram run-length encoding. O esquema da figura V-7 mostra como foram introduzidas as novas rotinas.

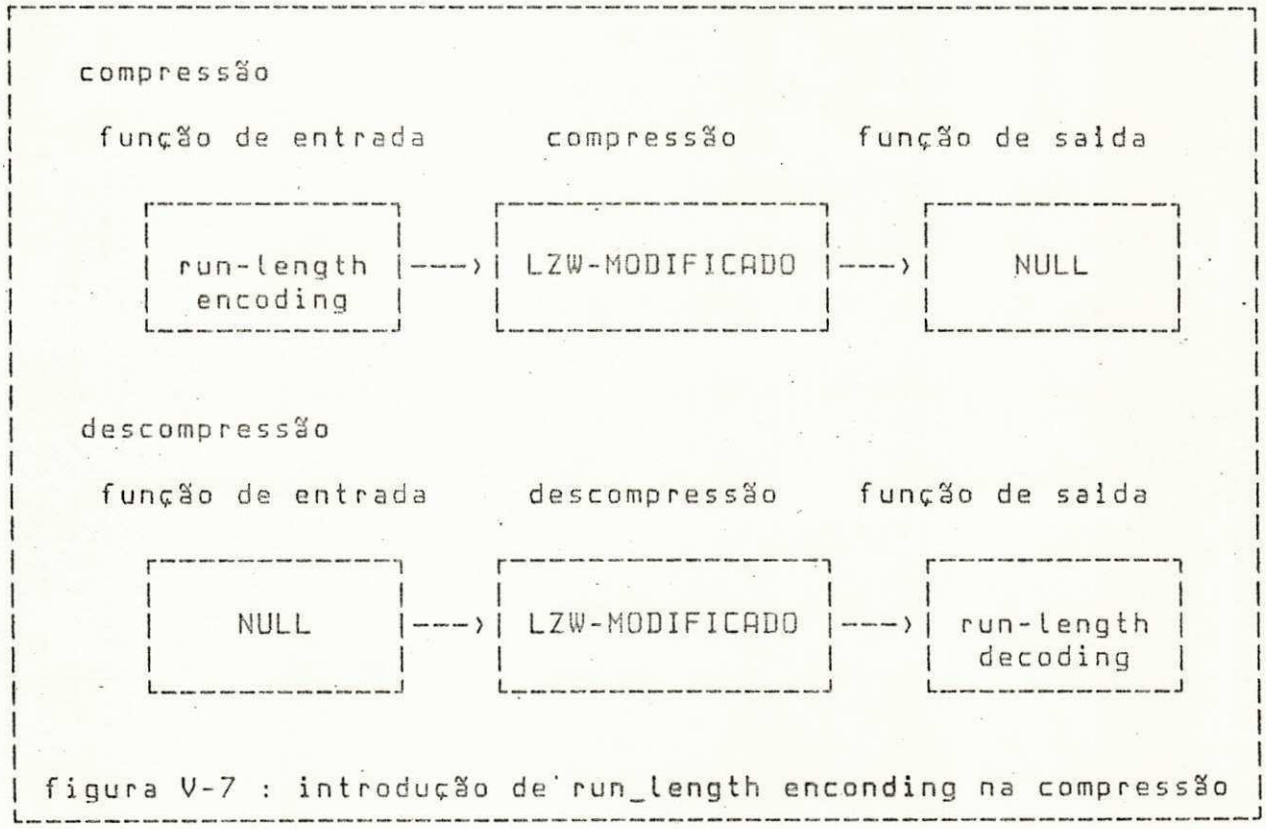


figura V-7 : introdução de run_length enconding na compressão

A figura V-8 apresenta comparações da velocidade de compressão e percentual de compressão alcançados pela versão otimizada

da biblioteca infocompress com e sem utilização de run-length encoding.

nome do arquivo	sem run-length encoding		com run-length encoding	
	velocidade	percentual	velocidade	percentual
apêndice	6911 c/s	46,86%	7079 c/s	47,30%
manual	10261 c/s	41,78%	8550 c/s	46,76%
iltd	9530 c/s	30,60%	7743 c/s	34,10%
c	10235 c/s	41,29%	7890 c/s	41,52%

figura V-8 : velocidade e percentual de compressão com e sem run-length encoding

Para os arquivos comprimidos a utilização de run-length encoding é responsável por um ganho médio de 13,98% no percentual de compressão atingido, em contrapartida a uma redução média de 5,60% na velocidade de compressão. A utilização de run-length encoding portanto, vai depender das necessidades da aplicação, ou seja, se o importante é um menor tempo de processamento ou um maior índice de redução (compressão).

CAPITULO VI

Conclusões e sugestões para continuação do trabalho

6.1 Conclusões

A modificação no algoritmo LZW aqui sugerida é bastante adequada para aplicações em que a informação a ser comprimida apresenta muita variação no tipo de redundância presente, permitindo que o algoritmo se readapte aos novos tipos de redundância encontrados. "Backup" de discos, por exemplo, envolve uma grande quantidade de informação apresentando geralmente características distintas, sendo portanto um exemplo prático da utilidade do algoritmo discutido.

A simplicidade do algoritmo, que permite implementações em hardware, também é outro aspecto que deve ser lembrado, além da característica de ser um algoritmo em "um passo", permitido portanto o uso em sistemas "on-line".

A utilização das funções da biblioteca infocompress que implementam o algoritmo LZW-MODIFICADO em um produto comercial, como o AGIX, e o papel importante que estas funções de compresão de dados tiveram no aumento da velociodade de transferência alcançado por este produto (veja figura V-5 do capitulo V), consolidam a contribuição dada por este trabalho à área de compressão de dados.

Vale ressaltar que um dos principais problemas existentes no algoritmo LZW se propaga no algoritmo LZW-MODIFICADO. Como

acontece com a maioria dos algoritmos adaptativos, a perda de sincronização entre emissor e receptor impossibilita a recuperação da mensagem original enviada. Este fato merece atenção especial quando se trata de transferência de informação comprimida em canais de comunicação suceptíveis a erros.

6.2 Sugestões para continuação do trabalho

Alguns pontos sobre o algoritmo LZW-MODIFICADO não foram ainda suficientemente explorados, e portanto podem ser alvo de futuros estudos. As seções seguintes dão alguns detalhes sobre estes pontos.

6.2.1 Utilização da capacidade de compressão instantânea do algoritmo como parâmetro para reinicialização da TS

Em alguns arquivos, a reinicialização da TS é responsável por uma queda considerável no percentual de compressão. Para estes arquivos, seria melhor que a TS não fosse reinicializada. Este fato ocorre quando a redundância do arquivo é bastante homogênea (veja o exemplo da figura V-4 do capitulo anterior) ou quando a reinicialização ocorre bem perto do término da operação de compressão, não permitindo uma reaprendizagem.

Uma alternativa para resolver este problema é o permanente monitoramento da capacidade de compressão do algoritmo [THOM 85]. Desta forma a TS só seria reinicializada se, estando a TS cheia, o algoritmo apresentasse uma queda significativa na capacidade de compressão em um determinado instante.

A utilização desse artifício pode gerar alguns problemas que devem ser observados com cuidado. Em primeiro lugar, a complexidade deste monitoramento permanente pode não ser aceitável para determinadas implementações (implementações em hardware por exemplo). Outro problema a ser resolvido é o da sincronização entre as operações de compressão e descompressão. É possível que seja necessário a definição de um código especial de sincronização, enviado pelo compressor, para avisar o descompressor que a TS foi reinicializada.

6.2.2 Estudo da complexidade do algoritmo e limites máximos de compressão

O objetivo deste estudo é se deter mais profundamente em detalhes da especificação e implementação dos algoritmos propostos. Artigos como [ZIV 77] e [BELL 86] entre outros, apresentam uma abordagem um tanto diferente. Preocupações com os limites mínimos e máximos dos percentuais de compressão, como também da complexidade dos algoritmos de compressão e descompressão, são discutidas mais detalhadamente nestas referências.

Uma análise assintótica do algoritmo LZW-MODIFICADO, aponta um índice de redução máximo bastante próximo do limite de 100%, quando o arquivo a ser comprimido apresenta certas características. Um esboço desta análise é apresentado a seguir.

Suponha a compressão de um arquivo formado exclusivamente por repetições de um mesmo caractere, digamos o caractere a, e uma TS com códigos de 12 bits. Para um arquivo formado por um único caractere, haveria, após o processo de compressão, uma

expansão do arquivo, pois um simbolo de 8 bits seria mapeado em um código de 12 bits. Para um arquivo formado por três caracteres o índice de redução seria nulo, já que os três caracteres de 8 bits seriam comprimidos em dois códigos de 12 bits. Pode-se notar que a medida que o arquivo vai crescendo, o índice de redução alcançado também cresce. No caso presente, o índice de redução seria máximo quando a TS enchesse. A tabela de 4096 códigos de 12 bits seria completamente preenchida quando 3840 (4096 - 256) códigos fossem gerados. A figura VI-1 ilustra este fato.

TAMANHO DO ARQUIVO	NUMERO DE CODIGOS	INDICE DE REDUÇAO
1	1	-50.00%
3	2	0.00%
6	3	25.00%
10	4	40.00%
7.374.720	3840	99.92%

figura VI-1 : IR máximo para arquivo contendo repetições de um único caractere

A relação entre o índice de redução e o tamanho do arquivo a ser comprimido é dada por:

$$IR = \frac{N_1 * 8 - i * 12}{N_1 * 8}$$

N_1 é a quantidade de caracteres a (tamanho do arquivo) necessários para que a operação de compressão do arquivo gere i

códigos e é dado por:

$N_i = \sum n$, n variando de 1 até i.

Quando a TS é reinicializada, o índice de redução sofre uma queda, como foi mostrado anteriormente, entretanto, com o repreenchimento da TS, o índice de redução volta a aumentar, até atingir o ponto máximo quando a TS se encontra novamente cheia.

O algoritmo LZW-MODIFCADO carece de um estudo mais aprofundado dos elementos acima citados, levando em consideração diferentes distribuições dos simbolos formadores dos arquivos fontes. Embora os valores apresentados no capitulo V, que representam medições em condições reais de operação, deixem claro os niveis de desempenho do algoritmo LZW-MODIFICADO, esforços no sentido de uma análise mais teórica, configurariam uma contribuição interessante para a área.

6.2.3 Implementações em hardware

A modificação sugerida pelo algoritmo LZW-MODIFICADO conserva a caracteristica de simplicidade na implementação do algoritmo LZW original. Welch em [WELC 84] propõe uma implementação em hardware bastante simples que apresenta uma velocidade de compressão muito alta.

A implementação do algoritmo LZW-MODIFICADO em hardware poderia ser feita com uma simples alteração no projeto proposto por Welch. Acreditamos que as velocidades atingidas por esta implementação seriam compativeis com aquelas obtidas por Welch.

Bibliografia

[ABRA 63] - N. Abramson, Information Theory and Coding, New York, McGraw-Hill, 1963.

[ALSB 75] - P. A. Alsberg, "Space and Time Savings Through Large Data Base Compression and Dynamic Restructuring", Proceedings of the IEEE, Vol. 63, No. 8, 1975, pag. 1114-1122.

[ASH 65] - R. B. Ash, Information Theory, New York, Interscience, 1965.

[BASS 85] - M. A. Bassiouni, "Data Compression in Scientific and Statistical Databases", IEEE Transactions on Software Engineering, Vol. SE-11, No. 10, 1985, pag. 1047-1058.

[BASS 86] - M. A. Bassiouni e B. Ok, "Double Encoding - A Technique for Reducing Storage Requirement of Text", Inform. Systems, Vol. 11, No. 2, 1986, pag. 177-184.

[BELL 86] - T. C. Bell, "Better OPM/L Text Compression", IEEE Transactions on Communications, Vol. Com-34, No. 12, 1986, pag. 1176-1182.

[BENT 86] - J. L. Bentley et al, "A Locally Adaptive Data Compression Scheme", Communications of ACM, Vol. 29, No. 4, 1986, pag. 320-330.

[BRAS 89a]- F. V. Brasileiro, J. A. B. Moura e J. P. Sauvé, "Compressão/Descompressão LZW Modificada", Anais do VII Simpósio Brasileiro de Redes de Computadores, Porto Alegre, Abril de 1989, pag. 649-664.

[BRAS 89b]- F. V. Brasileiro, J. A. B. Moura e J. P. Sauvé, "A Modified LZW Compression/Decompression Method", submetido à Computer.

[CLEA 84] - J. G. Cleary e I. H. Witten, "Data Compression Using Adaptive Coding and Partial String Matching", IEEE Transactions on Communications, Vol. Com-32, No. 4, 1984, pag. 396-402.

[COVE 73] - T. M. Cover, "Enumerative Source Encoding", IEEE Transactions on Information Theory, Vol. IT-19, No. 1, 1973, pag. 73-77.

[ELIA 75] - P. Elias, "Universal Codeword Sets and Representations of the Integers", IEEE Transactions on Information Theory, Vol. IT-21, No. 2, 1975, pag. 194-203.

[FALL 73] - N. Faller, "An Adaptive System for Data Compression", Record of the 7th Asilomar Conference on Circuits, Systems and Computers, Monterey, 1973, pag. 593-597.

[FANO 49] - R. M. Fano, Transmission of Information, Cambridge, M.I.T. Press, 1949.

[GALL 78] - R. G. Gallager, "Variation's on a Theme by Huffman", IEEE Transactions on Information Theory, Vol. IT-24, No. 6, 1978, pag. 668-674.

[GUAZ 80] - M. Guazzo, "A General Minimum-Redundancy Source-Coding Algorithm", IEEE Transactions on Information Theory, Vol. IT-26, No. 1, 1980, pag. 15-25.

[HAMA 86] - D. W. Hamaker, "Data Compression", Communications of ACM, Vol. 29, No. 3, 1986, pag. 173.

[HUFF 52] - D. A. Huffman, "A Method for the Construction of Minimum Redundancy Codes", Proceedings of de IRE, Vol. 40, No. 9, 1952, pag. 1098-1101.

[JONE 88] - D. W. Jones, "Application of Splay Trees to Data Compression", Communications of the ACM, Vol. 31, No. 8, 1988, pag. 996-1007.

[KANG 77] - A. N. C. Kang et al, "Storage Reduction Through Minimal Spanning Trees and Spanning Forests", IEEE Transactions on Computers, Vol. C-26, No. 5, 1977, pag. 425-434.

[KNUT 73] - D. E. Knuth, The Art of Computer Programming, Vol. 3 - Sorting and Searching, Reading, Addison-Wesley, 1973.

[KORN 73] - J. Korner e G. Longo, "Two-Step Encoding for Finite Sources", IEEE Transactions on Information Theory, Vol. IT-19, No. 6, 1973, pag. 778-782.

[LAES 86] - R. P. Laeser et al, "Engineering Voyager 2's Encounter with Uranus", Sci. Am., 255, 5, 1986, pag. 36-45.

[LANG 83] - G. G. Langdon, Jr. e J. J. Rissanen, "A Double-Adaptive File Compression Algorithm", IEEE Transactions on Communications, Vol. COM-31, No. 11, 1983, pag. 1253-1255.

[LANG 84] - G. G. Langdon, Jr., "An Introduction to Arithmetic Coding", IBM J. Res. Develop., Vol. 28, No. 2, 1984, pag. 135-149.

[LELE 87] - D. A. Lelewer e D. S. Hirschberg, "Data Compression", ACM Computing Surveys, Vol. 19, No. 3, 1987, pag. 261-296.

[MCIN 85] - D. R. McIntyre e M. A. Pechura, "Data Compression Using Static Huffman Code-Decode Tables", Communications of the ACM, Vol. 28, No. 6, 1985, pag. 612-616.

[PASC 76] - R. Pasco, "Source Coding Algorithm for Fast Data Compression", Dissertação de Ph.D., Departamento de Engenharia Elétrica, Universidade de Stanford, Stanford, 1976.

[POUN 87] - D. Pountain, "Run-Length Encoding", Byte, Junho de 1987, pag. 317- 320.

[PURS 77] - M. B. Pursley e K. M. Mackenthun, Jr., "Variable-Rate Coding for Classes of Sources with Generalized Alphabets", IEEE Transactions on Information Theory, Vol. IT-23, No. 5, 1977, pag. 592-597.

[REGH 81] - H. K. Reghbatti, "An Overview of Data Compression Techniques", Computer, Vol. 14, No. 4, 1981, pag. 71-76.

[RISS 76] - J. Rissanen, "Generalized Kraft Inequality and Arithmetic Coding", IBM J. Res. Develop., Vol. 20, 1976, No. 5, 1976, pag. 198-203.

[RISS 79] - J. Rissanen e G. G. Langdon, Jr., "Arithmetic Coding", IBM J. Res. Develop., Vol. 23, No. 2, 1979, pag. 149-162.

[RUBI 79] - F. Rubin, "Arithmetic Stream Coding Using Fixed Precision Registers", IEEE Transactions Information Theory, Vol. 25, No. 6, 1979, pag. 672-675.

[RUTH 72] - S. S. Ruth et al, "Data Compression for Large Business Files", Datamation, Vol. 18, No. 9, 1972, pag. 62-66.

[SCIA 87] - R. J. Sciamanda, "Another Approach to Data Compression", Byte, fevereiro de 1987, pag. 137-140.

[SHAN 49] - C. E. Shannon e W. Weaver, The Mathematical Theory of Communications, Urbana, University of Illinois Press, 1949.

[STOR 82] - J. A. Storer e T. G. Szymanski, "Data Compression via Textual Substitution", Journal of the ACM, Vol. 29, No. 4, 1982, pag. 928-951.

[TANA 87] - H. Tanaka, "Data Structure of Huffman Codes and its Application to Efficient Encoding and Decoding", IEEE Transactions Information Theory, Vol. 33, No. 1, pag. 154-156.

[THOM 85] - S. W. Thomas et al, implementação de dominio público do algoritmo LZW.

[VITT 87] - J. S. Vitter, "Design and Analysis of Dynamic Huffman Codes, Journal of the ACM, Vol. 34, No. 4, 1987, pag. 825-845.

[WELC 84] - T. A. Welch, "A technique for high performance data compression", Computer, Vol. 17 No. 6, 1984, pag. 8-19.

[WHIT 67] - H. E. White, "Printed English Compression by Dictionary Encoding", Proceedings of the IEEE, Vol. 55, No. 3, 1967, pag. 390-396.

[WITT 87] - I. H. Witten et al, "Arithmetic Coding for Data Compression", Communications of the ACM, Vol. 30, No. 6, 1987, pag. 520-540.

[ZIV 77] - J. Ziv e A. Lempel, "A universal Algorithm for Sequential Data Compression", IEEE Transactions on Information Theory, Vol. IT-23, No. 3, 1977, pag. 337-343.

[ZIV 78] - J. Ziv e A. Lempel, "Compression of Individual Sequences via Variable-Rate Coding", IEEE Transactions on Information Theory, Vol. IT-24, No. 5, 1978, pag. 530-536.