Métodos Universais de Compressão de Dados

Este exemplar corresponde à redação final da tese devidamente corrigida e defendida pela Srta. Fabíola Gonçalves Pereira de Souza e aprovada pela Comissão Julgadora.

Campinas, 16 de dezembro de 1991.

Prof. Dr. Cláudio Leonardo Lucchesi.

Orientador

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

Métodos Universais de Compressão de Dados[1]

Fabíola Gonçalves Pereira de Souza[2]

Departamento de Ciência da Computação
IMECC - UNICAMP

Banca Examinadora:

- Cláudio Leonardo Lucchesi (Orientador)[3]
- Jorge Stolfi[4]
- Tomasz Kowaltowski[3]
- Paulo Lício de Geus (Suplente)[3]

[1]Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação da UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

[2]A autora é Bacharela em Processamento de Dados pela Universidade Federal da Bahia.

[3]Professor do Departamento de Ciência da Computação - IMECC - UNICAMP.

[4]DEC Systems Research Center - Palo Alto, Califórnia.

Aos meus pais —
uma experiência sem fim que me foi dada

Agradecimentos

Ao CNPq e à FAPESP pelo apoio financeiro recebido.

A todos os colegas e professores do DCC, que indiretamente, por vezes decididamente, auxiliaram no desenvolvimento deste trabalho. Em particular, ao colega Armando, que muito prestativamente encontrou os compressores comerciais aqui utilizados.

A Leila, que me privilegiou com algo mais duradouro do que a sua excelente companhia nos estudos, a amizade.

A minhas irmãs e meus amigos da Bahia, que sempre estiveram tão presentes durante a realização deste trabalho.

A minha mãe, pelo seu constante apoio e sabedoria.

Ao Prof. Cláudio L. Lucchesi, pela confiança, pela precisa e excelente orientação.

A Ronney, por viver minha época e refletir comigo os muitos momentos... uma gratidão que desafia o tempo e a distância.

" Lewis Carroll(...) observa na segunda parte do extraordinário romance onírico Sylvie and Bruno *— ano de 1893 — que sendo limitado o número das palavras que compreende um idioma, também é o das suas combinações, ou seja, o dos seus livros. "Logo", diz, "os literatos não se perguntarão Que livro escreverei?mas Qual livro?"(...)*

A idéia básica de Lasswitz é a de Carroll, mas os elementos do seu jogo são os universais símbolos ortográficos, não as palavras de um idioma. O número de tais elementos — letras, espaços, parênteses, reticências, algarismos — é reduzido e pode se reduzir ainda mais. O alfabeto pode renunciar ao q (que é totalmente supérfluo), ao x (que é uma abreviatura) e a todas as letras maiúsculas. Podem se eliminar os algarismos do sistema decimal de numeração ou se reduzir a dois, como na notação binária de Leibniz. Pode-se limitar a pontuação à vírgula e ao ponto. Pode não haver acentos, como no Latim. À base de simplificações análogas, Kurd Lasswitz chega a vinte e cinco símbolos suficientes (vinte e duas letras, o espaço, o ponto, a vírgula) cujas variações com repetição abarcam tudo que é dado expressar: em todas as línguas."

Jorge Luis Borges
"A Biblioteca Total"

Resumo

A Compressão de Dados objetiva representar os dados de maneira reduzida. Este trabalho apresenta os principais métodos de compressão para dados textuais, cuja compressão exibida seja universal (a compressão se adapta a qualquer tipo de dado) e fiel (os dados podem ser recuperados integralmente).

A dissertação consiste de quatro Capítulos. O Capítulo 1 apresenta o assunto.

O Capítulo 2 introduz conceitos necessários ao entendimento dos métodos. Desta maneira, realiza uma caracterização dos códigos adotados na representação; esclarece porque é possível reduzir o comprimento dos textos; engloba os métodos numa classificação.

O Capítulo 3 apresenta os métodos. Para tanto, realiza uma descrição do processo de compressão adotado, fornece uma avaliação teórica do seu desempenho, e detalha possíveis implementações. Os métodos estudados são: Shannon-Fano, Huffman, Aritmético, Elias-Bentley e Lempel-Ziv. Além disso, sempre que possível, é feita uma associação entre os métodos mencionados e os utilitários: PKPAK, PKZIP, ICE, LHA, ARJ e também *pack, compact* e *compress.*

O Capítulo 4 apresenta uma avaliação empírica sobre o desempenho dos vários métodos estudados, bem como uma comparação que também envolve os utilitários mencionados.

Abstract

The objective of Data Compression is to reduce the size of data representation. In this work we present the most important methods for universal and lossless compression, that is, methods which are applicable to all kinds of data and allow full recovery of information.

This dissertation consists of four chapters. In Chapter 1 we present the subject.

In Chapter 2 we introduce the fundamental concepts in data compression. Thus, we characterize codes, redundancy and classify data compression methods.

In Chapter 3 we describe the most important methods and discuss their performance and possible implementations; the methods are: Shannon-Fano's, Huffman's, Arithmetic Coding, Elias-Bentley's and Lempel-Ziv's. In addition, whenever possible, an association is made with well-known data compression programs such as PKPAK, PKZIP, ICE, LHA, ARJ and *pack, compact* and *compress.*

In Chapter 4 we give an empirical evaluation of the performance of those methods, as well as a comparison with the well-known programs mentioned above.

Conteúdo

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Neste Capítulo, introduzimos o conceito de compressão de dados e caracterizamos o tipo de compressão que pretendemos estudar. Apresentamos também um resumo dos tópicos a serem abordados no restante deste trabalho.

1.1 Apresentação

Muitas aplicações que trabalham com informações que requerem grande volume de dados têm surgido em meios desprovidos de grande capacidade de armazenamento. A utilização de bibliotecas de recursos gráficos, imagens sintéticas, dicionários de termos léxicos e ortográficos, e de muitos outros dados têm sido uma constante em computadores pessoais. O seu uso, na maior parte das vezes, depende de uma economia de espaço.

Por outro lado, o crescente uso de redes de comunicação de dados, seja a nível local ou não, tem aumentado enormemente o volume de transferência de informações sobre os canais de comunicações. Tal fato tem requerido uma maior velocidade de transmissão dos dados e conseqüentemente uma maior economia no tempo de transmissão.

A economia de espaço e de tempo necessária nas áreas de armazenamento de dados e de transmissão de dados, respectivamente, torna-se viável a partir de um compressão dos dados. A compressão permite uma redução dos custos de armazenamento e custos de comunicação. Isto porque ela aumenta a capacidade do dispositivo de armazenamento ou do canal de comunicação. Se um arquivo é comprimido em 50% do seu tamanho, tem-se o efeito de se duplicar a capacidade do dispositivo onde este será armazenado ou de duplicar a velocidade de transmissão. O que se observa é que a compressão é especialmente útil às áreas de armazenamento de dados e de transmissão de dados, e seu uso torna-se não só uma opção, mas uma necessidade.

Afora o interesse prático, a compressão é uma área que desperta interesses teóricos. Questões relativas à síntese e análise dos seus métodos e ao entendimento da máxima compressão possível exibem problemas de boa complexidade matemática. Lelewer e Hirschberg [LH87, p. 268] ainda apresentam uma lista de aplicações em outras áreas, e que são derivadas de algoritmos para compressão de dados; por exemplo, a partir do método de Huffman [Huf52] obtém-se um algoritmo eficiente para a determinação de árvores ótimas de busca.

A Compressão de Dados tem por preocupação a representação do dado de maneira reduzida. Sendo assim, o efeito de comprimir um arquivo de dados é o da redução do tamanho deste arquivo. Os arquivos que estão numa representação particular, tais como ASCII, EBCDIC, binário puro, octal, hexadecimal, *bitmap*, etc. são transformados em arquivos numa representação comprimida. Futuramente, o arquivo original necessitará ser recuperado, e isto é feito a partir da descompressão. O processo de compressão fica bem expresso a partir do esquema da figura 1.1, e o que se percebe é que ele vem a acrescentar mais uma hierarquia na representação dos dados.

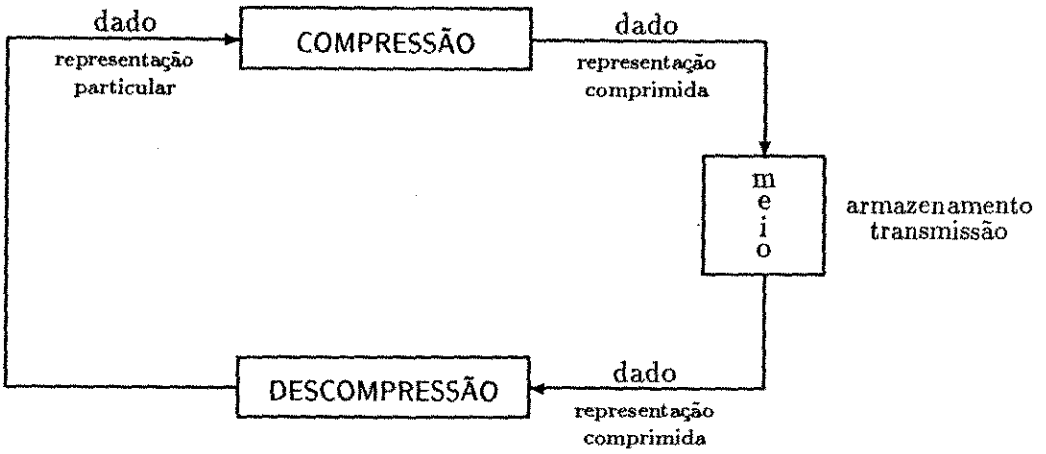


Figura 1.1: **Processo de Compressão.**

A compressão pode ser implementada tanto em *hardware* como em *software*, o que exige uma capacidade adicional de processamento. Assim, um bom compressor não só deve possibilitar uma alta taxa de redução dos arquivos como também deve realizá-la com bastante rapidez, sendo este um fator crítico na comunicação de dados. Para avaliar a capacidade de compressão estaremos adotando a seguinte medida:

$$T(\%) = (1 - \frac{C}{F}) * 100 \tag{1.1}$$

onde T é a *taxa de compressão*, C é o comprimento do arquivo comprimido e F é o comprimento do arquivo fonte. Esta medida expressa a quantidade de compressão alcançada em relação ao tamanho do arquivo original; assim, quanto mais perto de 100 estiver, maior terá sido a compressão realizada.

A recuperação de um arquivo comprimido tanto pode ser fiel, isto é, os dados são recuperados tais como eram, quanto pode ser aproximada, neste caso, é recuperada apenas uma aproximação do dado. No primeiro caso, de compressão *fiel*, a informação contida no dado não é perdida, sendo conservada, diferentemente do segundo caso, onde a compressão é de *perdas*. O tipo de compressão a ser escolhida depende da natureza da aplicação. Para aplicações com *dados textuais*, que são textos em línguas naturais, códigos em linguagem fonte, códigos executáveis, dados numéricos, dados gráficos, imagens sintéticas, etc. é usada

a compressão fiel. Assim, desejando-se comprimir, por ex., a letra "A", só tem sentido se recuperar integralmente a letra "A" e não uma aproximação desta. Para dados analógicos digitalizados de som, fala, música, imagem, vídeo, fotografia, e semelhantes é usada a compressão de perdas. Neste caso é admissível a perda de alguma característica irrelevante do dado. Alguns autores designam esta compressão de perdas por *compactação*. No presente estudo, estaremos interessados na compressão fiel, onde há uma conservação da informação que o dado apresenta.

A compressão compartilha com outras áreas a preocupação da representação dos dados. Enquanto esta visa comprimi-los, outras, como a Criptografia, visam sua segurança; já a detecção/correção de erros, visa sua integridade. Alguns desses objetivos são antagônicos, enquanto outros se auxiliam. A interação entre a compressão e essas diversas áreas não será nosso objeto de estudo. Sabe-se, entretanto, que os dados comprimidos estarão sujeitos a ruídos nos meios de armazenamento ou comunicação e tem-se que nem todos os códigos de compressão existentes apresentam a habilidade de permitir correção de erros, sendo que muitos são bastante susceptíveis à sua propagação. A tolerância ou não ao erro vai depender das características do código. Este vem a ser um problema crítico no que se refere a sistemas de comunicação. Apesar disto, iremos considerar que os dados numa representação comprimida não poderão ser danificados e serão recuperados fielmente, o que significa que estaremos supondo um meio de armazenamento ou comunicação sem ruído.[1]

Existem diversas maneiras de se efetuar compressão, e isto é feito através de métodos de compressão que apresentam algoritmos de compressão e descompressão. Alguns dos métodos são específicos para certos tipos de dados, neste caso são ditos *métodos semânticos*, outros se pretendem genéricos e são ditos *métodos universais*. Seus algoritmos se caracterizam pela simplicidade e rapidez com que efetuam a compressão e a descompressão. Muitos são métodos em linha (*on-line*), o que permite que a descompressão possa ser realizada concomitantemente à compressão, sendo esta uma característica essencial para sistemas de comunicação. Vários dos métodos têm sido usados por aplicações particulares; Cormack [Cor85] e Klein *et alli* [KBD89], por exemplo, apresentam métodos eficientes para recuperação de informação em bases de dados comprimidas. Outros métodos compõem ferramentas de uso genérico. Algumas destas ferramentas têm sido bastante empregadas, tais como os utilitários *compress, pack,* e *compact* presentes no sistema UNIX[2] e as ferramentas PKPAK, PKZIP, ICE, LHA e ARJ co-

[1] Em inglês, *noiseless channel*.

[2] UNIX é uma marca registrada dos laboratórios Bell da AT&T.

nhecidas dos usuários de computadores pessoais e de redes de compartilhamento de dados. Nestes compressores, a depender do tipo de arquivo fonte, a redução pode ser dramática, atingindo valores superiores a 80% de compressão em relação ao tamanho do arquivo original. As melhores taxas de compressão obtidas variam em torno de 55% para código executável, 65% para textos em linguagem natural, 70% para código fonte e 80% para imagens (no formato *bitmap*).

Os métodos semânticos têm sua construção adequada a certos tipos de dados. Podemos citar como exemplo a *eliminação de caracteres repetidos*, também conhecida como *codificação por carreiras*[3], onde normalmente os caracteres que se repetem são substituídos por um código representando um contador e o caractere. Existe a *codificação por diferenças*, onde os dados, geralmente numéricos, guardam uma relação entre si e são representados como diferenças em relação ao valor do dado anterior. Existe a *substituição por dicionário*, onde padrões que sempre aparecem são substituídos por índices a uma única cópia existente num dicionário. Há a *notação compactada*, como o próprio tipo *pack* de algumas linguagens de programação. Enfim, existem muitas dessas abordagens. Algumas delas têm sido usadas para a compressão de imagens, como é o caso das duas primeiras,[LH87] e a maioria têm aplicação em bases de dados. Em verdade, grande parte pode ser vista como casos particulares dos métodos universais; é o que acontece com a substituição por dicionário, ou a codificação por carreiras. Apesar da especificidade, muitos desses métodos semânticos são usados, em parte pela simplicidade e também porque são bem eficazes para certos tipos de dados, como é o caso da eliminação de caracteres repetidos para imagens; por outro lado, a maioria serve para eliminar alguma redundância na representação do dado já no momento da sua geração.

Os métodos em que estamos interessados são os métodos universais de compressão. Eles são construídos com a suposição do não conhecimento acerca dos dados que serão comprimidos. O arquivo original é visto como um *seqüência de bits*, e o método deverá descobrir alguma característica na estrutura apresentada que permita a compressão. Escolhemos exemplos significativos de métodos para apresentar como eles costumam atuar sobre os dados. Procuramos também englobá-los em categorias, já que percebemos que muitos, na sua tentativa de compressão, seguem um mesmo caminho.

Apresentamos na figura 1.2 uma caracterização da compressão que pretendemos estudar no restante deste trabalho. Esta caracterização ficou estabelecida a partir dos requisitos impostos à compressão e que foram expostos nesta seção.

A seguir, apresentamos os tópicos a serem abordados neste estudo.

[3] Do inglês, *run-length encoding*.

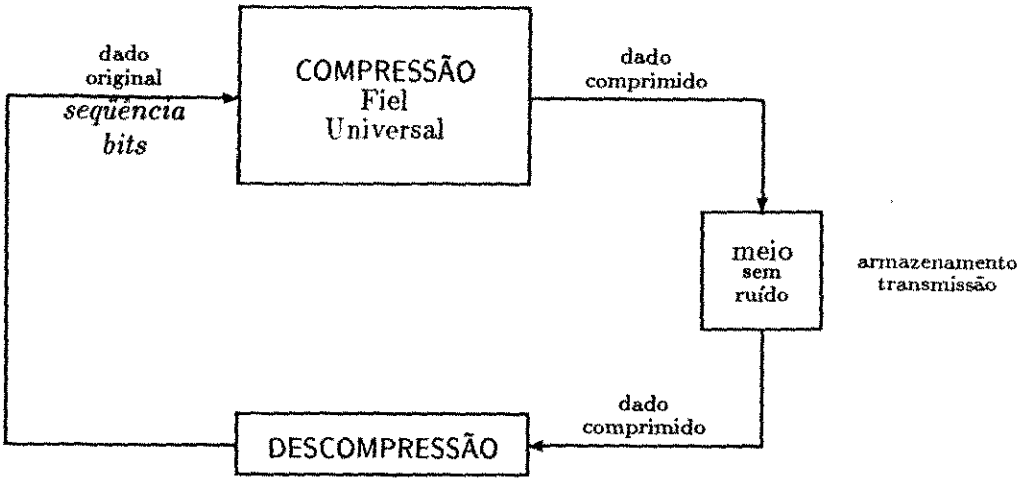


Figura 1.2: **Compressão a ser estudada.**

1.2 Conteúdo

No Capítulo 2 deste estudo introduzimos conceitos importantes para o entendimento da compressão. Em verdade, fornecemos algumas ferramentas para a construção de métodos de compressão. Apresentamos o conceito de *código* e os requisitos que estes deverão ter quando aplicados à compressão. Esclarecemos porque é possível reduzir o tamanho dos textos e finalmente estabelecemos diferenças entre os métodos, o que nos permite englobá-los numa classificação.

O Capítulo 3 apresenta nas duas primeiras seções diversos métodos de compressão. Escolhemos para apresentar aqueles importantes na sua categoria e que têm seu desempenho destacado, não só na comunidade acadêmica, mas na composição de aplicativos práticos. Estes métodos são o pioneiro método de Shannon-Fano [SW49] [Fan49], o clássico método de Huffman [Huf52] [Fal73] [Gal78] [Knu85], o elegante método Aritmético [Abr63] [WNC87], o inovativo método de Lempel-Ziv [ZL77] [ZL78] e o recente método de Elias-Bentley *et alii* [BSTW86] [Eli87]. Alguns desses métodos, por sua vez, despertaram em outros autores novas idéias. Portanto, algumas variações desses métodos também serão apresentadas. A última seção do Capítulo 3 apresenta alguns dos mais destacados compressores genéricos, comerciais ou de livre distribuição, que têm por base os métodos apresentados. Estes compressores são o *pack*, *compact*, *compress*,

PKPAK, PKZIP, ICE, LHA e ARJ.

O Capítulo 4 realiza uma comparação, a nível de taxa de compressão, entre os métodos apresentados, que foram implementados, e entre os compressores genéricos. A comparação levou em conta tipo e comprimento dos arquivos.

Capítulo 2

Conceitos

Neste Capítulo, apresentamos conceitos importantes para o entendimento dos métodos de compressão. Tendo por base a Teoria da Codificação, a seção 2.1 caracteriza os códigos para a compressão. A seção 2.2, principalmente a partir da Teoria da Informação, apresenta algumas questões teóricas e esclarece porque é possível reduzir o comprimento dos textos. Finalmente, a seção 2.3 propõe uma classificação para os métodos de compressão que tem por base as características desejadas a um processo de compressão ressaltadas nas seções anteriores.

2.1 Códigos para Compressão

A compressão foi vista como um processo de codificação que transforma um texto fonte, gerado em uma representação qualquer, em um texto comprimido, a partir de uma representação comprimida. A codificação que se efetua na compressão obedece a alguns requisitos que passamos a expor.

Tanto o texto fonte como o texto comprimido estão expressos numa representação binária. Denota-se por Σ o alfabeto binário $\{0,1\}$.

Definição 2.1 [Código] O *alfabeto de símbolos fonte* S e o *alfabeto de códigos* C são partes finitas de Σ^*. Um *código* δ é uma função que associa símbolos do alfabeto fonte S a símbolos do alfabeto de códigos C. Tem-se então,

$$\delta : S \mapsto C$$

Convenciona-se ainda que δ aplicada a uma cadeia em S^* é a concatenação de δ aplicada individualmente aos símbolos da cadeia. Isto é:

$$\delta(s_1 s_2 \ldots s_k) = \delta(s_1)\delta(s_2)\ldots\delta(s_k) \quad k \geq 0, \quad s_i \in S,\ 1 \leq i \leq k$$

Iremos ilustrar melhor o conceito de código a partir do seguinte exemplo.

Exemplo 2.2 Seja $S := \{o, foi, compresso, texto\}$ e $C := \{0, 10, 110, 111\}$. Um código δ pode ser:

$$\begin{aligned} \delta(o) &= 0 \\ \delta(foi) &= 10 \\ \delta(compresso) &= 110 \\ \delta(texto) &= 111 \end{aligned}$$

Para o texto fonte: *o texto foi compresso*, a codificação será:

$$\delta(o\ texto\ foi\ compresso) = 0\,111\,10\,110.^{1}$$

Convém esclarecer que a definição 2.1 é relativamente simplista; alguns dos códigos a serem apresentados não se encaixam exatamente nesta definição; entretanto, uma formalização mais abrangente seria de difícil leitura e inapropriada neste ponto.

[1] Estaremos utilizando espaços em branco entre os *bits* dos códigos apenas para facilitar a leitura dos mesmos.

Estaremos designando o termo *código* tanto para a função δ como para a sua imagem, isto é, o código obtido para a cadeia em S^*. Assim, no ex., não somente δ é um código, mas também 011110110 é o código para o texto fonte específico. A distinção entre eles ficará implícita pelo contexto.

Uma vez que a compressão que se está estudando é fiel, não é possível mapear dois símbolos diferentes em códigos iguais, já que a informação contida nos símbolos não é a mesma. Portanto, um dos requisitos para o código é que este seja *injetor*.

Os símbolos em S e os códigos em C podem ter comprimento fixo ou variável em Σ^*.

De acordo com o comprimento apresentado pelos símbolos fonte e símbolos de código, o código é classificado, respectivamente, em *fixo-fixo, variável-fixo, fixo-variável ou variável-variável.*

Nos exemplos que se seguem, as letras romanas indicarão símbolos do alfabeto fonte, cuja representação binária tem comprimento fixo. O próximo exemplo irá ilustrar então um código do tipo fixo-variável.

Exemplo 2.3 Seja $S := \{a, b, c, d\}$ e $C := \{1, 10, 01, 1100\}$. Um código δ pode ser:

$$\begin{aligned} \delta(a) &= 1 \\ \delta(b) &= 10 \\ \delta(c) &= 01 \\ \delta(d) &= 1100 \end{aligned}$$

Com o código δ obtém-se, a partir do exemplo de texto fonte: *abcda*, o seguinte texto comprimido:

$$\delta(abcda) = 1\,10\,01\,1100\,1.$$

Observa-se no exemplo anterior que existem outros textos fonte que darão origem ao mesmo texto comprimido, pois $\delta(abcda) = \delta(daabc) = \delta(dada)$. Portanto, apenas a restrição do código ser injetor não é suficiente para que não haja ambigüidades na recuperação do texto fonte, ou na decodificação a partir do texto comprimido. Esta ambigüidade não é desejável, já que a compressão é universal e excluiu-se qualquer conhecimento semântico sobre o texto fonte.

O requisito que se impõe para a função de codificação é que o código por ela gerado seja unicamente decodificável.

Definição 2.4 [Código Unicamente Decodificável] Um código δ é *unicamente decodificável* se qualquer cadeia em C^*, na imagem de δ, puder ser decodificada em

uma única cadeia em S^*. Ou seja, para uma cadeia $\theta \in C^*$, existe no máximo uma cadeia $\alpha \in S^*$, tal que $\delta(\alpha) = \theta$. Isto significa que:

$$\delta : S^* \mapsto C^*$$

deve ser injetora.

Para uma ilustração do conceito, considere o exemplo que segue:

Exemplo 2.5 Seja $S := \{a, b, c\}$ e $C := \{1, 100, 000\}$. Um código δ, unicamente decodificável, pode ser:

$$\begin{aligned} \delta(a) &= 1 \\ \delta(b) &= 100 \\ \delta(c) &= 000 \end{aligned}$$

Com este código, qualquer cadeia em C^* corresponde a no máximo uma única cadeia em S, tal como: $100\,000\,000\,1 = \delta(bcca)$.

Algoritmos eficientes que determinam se um código é unicamente decodificável são apresentados por Even [Eve79], e Apostolico e Giancarlo [AG84].

Uma vez que os códigos comprimidos deverão obedecer as restrições até aqui impostas de serem injetores e unicamente decodificáveis, iremos convencionar que sempre que nos referirmos a *código* este deverá ter estas características.

Apesar da restrição para o código ser injetor e unicamente decodificável, a decodificação nem sempre é feita de maneira imediata, ou seja, após a leitura do código, nem sempre se consegue determinar de imediato qual o símbolo correspondente a este. No exemplo anterior, isto pode ser observado; o código de b contém o de a, ocasionando incerteza se o texto fonte começa por ac ou por b. Isto faz com que a decodificação de uma boa parte do texto seja adiada até que se consiga determinar exatamente os símbolos correspondentes, o que é feito com base na quantidade de zeros; se a quantidade de zeros é múltipla de três, então o texto se inicia por ac, de outra forma por b.

A restrição que se impõe para códigos que requerem uma instantaneidade na decodificação, ou mais precisamente, uma *decodificação em tempo real*, em tempo proporcional ao tamanho do código para cada símbolo, é que estes sejam de prefixo.

2.1.1 Códigos de Prefixo

Definição 2.6 [Código de Prefixo] Um código é dito de *prefixo* se nenhum código em C é prefixo próprio de outro código em C.

O exemplo que segue apresenta um código de prefixo.

Exemplo 2.7 Seja $S := \{a, b, c, d, e\}$ e $C := \{000, 010, 100, 101, 11\}$. Um código δ, de prefixo, pode ser:

$$\begin{aligned} \delta(a) &= 11 \\ \delta(b) &= 100 \\ \delta(c) &= 000 \\ \delta(d) &= 010 \\ \delta(e) &= 101 \end{aligned}$$

Observa-se que nenhum código para os símbolos é prefixo de outro, e assim, por ex., o texto :

$$100\,000\,000\,010\,101\,11\,000\,11\,010\,010 = \delta(bccdeacadd)$$

poderá ser decodificado em tempo real.

Quando o código é do tipo fixo-fixo ou variável-fixo, este necessariamente é de prefixo, pois uma vez que o comprimento para os códigos é fixo, se existissem códigos que fossem prefixos de outros, estes seriam iguais, mas isso não pode acontecer já que considera-se que o código é injetor.

A seguir, apresentamos um conjunto de códigos de prefixo para inteiros positivos proposto por Elias [Eli75], cuja descrição também é fornecida por Lelewer e Hirschberg [LH87]. Nestes códigos é associado a cada número um código binário de comprimento variável. Quanto maior o valor do número maior será o comprimento do código.

Codificação de Elias para Inteiros O primeiro código de Elias, designado por código γ, associa ao inteiro $i > 0$ a sua representação binária precedida por $\lfloor \log i \rfloor$ zeros. Por ex., o código para o número 33 será: 00000 100001, e conterá 11 *bits*. É fácil verificar que este código é de prefixo, pois a quantidade de zeros no início indica a quantidade de *bits* seguintes que formam o número. Após os zeros virá sempre o dígito 1 seguido de dígitos binários que ocupam o mesmo espaço dos zeros. Ao todo, o número será representado por:

$$1 + 2\lfloor \log i \rfloor \ \textit{bits}.$$

O código γ pode ser novamente aplicado à quantidade $\lfloor \log i \rfloor + 1$, que representa os zeros do prefixo juntamente com o *bit* 1, originando o código δ para i. Este código δ é formado por $\gamma(\lfloor \log i \rfloor + 1)$ mais a representação binária de i, sem o *bit* 1 inicial. Por ex., $\delta(33) = \gamma(6)00001 = 00110\,00001$, e conterá 10 *bits*. Ao todo, o código δ representa o número por:

$$1 + \lfloor \log i \rfloor + 2\lfloor \log(1 + \log i) \rfloor \ bits.$$

O código γ pode mais uma vez ser aplicado para a quantidade $\lfloor \log(1+\log i) \rfloor + 1$. Por ex., para o número 33, a codificação será: $\gamma(3)10\,00001 = 011\,10\,00001$, e conterá 10 *bits*. Este processo pode ser continuado, entretanto observa-se que esta continuação só é recomendável quando os valores numéricos são muito grandes.

Na geração de códigos algumas otimizações podem ser realizadas. Uma delas é procurar diminuir o comprimento dos códigos gerados.

Definição 2.8 [Código de Prefixo Minimal] Um código de prefixo é *minimal* se tomando-se c como um prefixo próprio de um código em C, então $c0$ e $c1$ existem como códigos ou prefixos próprios de códigos.

Para uma ilustração do conceito, considere o exemplo que segue:

Exemplo 2.9 Seja $S := \{a, b, c, d, e\}$ e $C := \{00, 01, 100, 101, 11\}$. Um código δ, de prefixo minimal, pode ser:

$$\begin{aligned} \delta(a) &= 11 \\ \delta(b) &= 100 \\ \delta(c) &= 00 \\ \delta(d) &= 01 \\ \delta(e) &= 101 \end{aligned}$$

A codificação para o mesmo texto do exemplo anterior será:

$$\delta(bccdeacadd) = 100\,00\,00\,01\,101\,11\,00\,11\,01\,01,$$

menor que a do exemplo anterior, que apresenta código de prefixo mas não minimal.

Observa-se que a minimalidade evita que se gerem códigos com *bits* desnecessários.

Os códigos de prefixo correspondem naturalmente a estruturas do tipo árvore. A seguir apresentaremos as árvores de prefixo com suas propriedades. Este estudo é importante em virtude da maioria dos métodos apresentarem códigos de prefixo minimal.

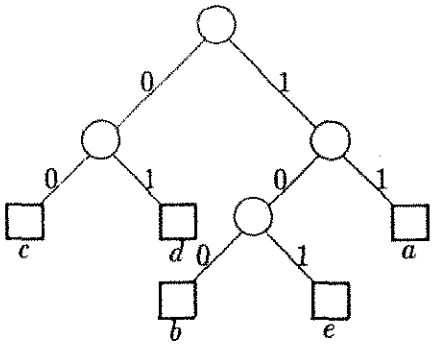


Figura 2.1: Árvore de prefixo minimal para o exemplo 2.9.

Árvore de Prefixo

Observe a árvore da figura 2.1. Seus ramos estão rotulados pelos símbolos binários. Os nodos folhas estão rotulados por símbolos de um alfabeto determinado. Pode-se entender o percurso da raiz até os nodos folhas como códigos associados aos símbolos presentes nos nodos. Observa-se que esta árvore corresponde exatamente ao código δ do exemplo 2.9.

Definição 2.10 [Árvore de Prefixo] Uma *árvore de prefixo* é uma árvore binária em que os ramos esquerdos são rotulados pelo símbolo binário 0, os ramos direitos pelo símbolo binário 1 e os nodos folhas estão associados a símbolos do alfabeto fonte.

Os símbolos binários encontrados no percurso da raiz até uma folha corresponde ao código para o símbolo associado àquela folha. O tamanho do código será o tamanho do percurso, ou nível do símbolo na árvore.

É fácil verificar que esta construção corresponde a um código de prefixo, e que, reciprocamente, todo código de prefixo tem uma árvore de prefixo associada. No restante deste trabalho, iremos utilizar estes dois termos indistintamente.

A visualização de códigos de prefixo como uma árvore facilita o entendimento de muitos conceitos. Um deles é o da codificação/decodificação em tempo real. Para codificar um símbolo, basta gerar os *bits* que se encontram no caminho que vai da folha, onde está representado o símbolo, até a raiz. Como o código é obtido em ordem reversa, uma estrutura do tipo pilha deve ser utilizada num passo intermediário para a sua geração. Na decodificação de um texto da esquerda para a direita, basta percorrer-se, na árvore a partir da raiz, o caminho que os símbolos binários indicam; quando se alcançar a folha é porque decodificou-se

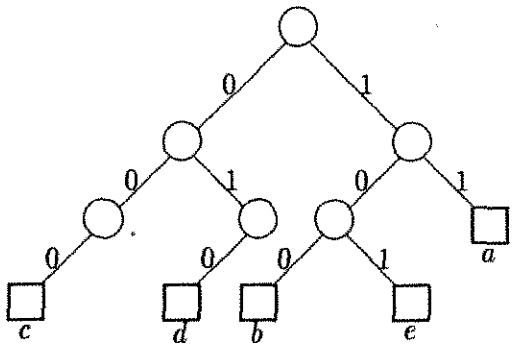


Figura 2.2: Árvore de prefixo para o exemplo 2.7.

um símbolo. Se tomarmos o texto codificado do exemplo 2.9 e realizarmos este procedimento na árvore da figura 2.1, iremos facilmente chegar ao texto fonte. Este processo de decodificação permite que se note que a restrição para que os símbolos do alfabeto fonte sejam nodos folhas fornece uma garantia para que o código seja de prefixo.

Outro conceito fácil de ser verificado é o da minimalidade. A árvore da figura 2.1 corresponde a um código de prefixo minimal. Observe a árvore da figura 2.2. Esta árvore equivale ao código do exemplo 2.7, que é de prefixo, mas não minimal. Nota-se que alguns dos nodos têm apenas um filho e podem, portanto, ser eliminados, conseqüentemente diminuindo o tamanho do código. Isto nos leva ao seguinte resultado.

Proposição 2.11 *Toda árvore de prefixo minimal é cheia.*

Define-se *árvore binária cheia* como aquela em que todo nodo possui zero ou dois filhos.

Alguns dos métodos de compressão a serem abordados necessitam representar no texto comprimido a tabela de códigos (símbolos com respectivos códigos) ou a árvore de prefixo para uma futura descompressão.

Para a representação da árvore de prefixo basta que se codifique apenas a sua forma (ou "esqueleto") seguida de uma codificação para os símbolos do alfabeto que obedeça a mesma ordem de codificação dos nodos na árvore.

A forma de uma árvore de prefixo cheia pode ser representada com $2|S| - 1$ *bits*, onde $|S|$ é o tamanho do alfabeto ou a quantidade de folhas na árvore [LH87]. Isto significa que é gerado um *bit* para cada nodo. Um algoritmo para tanto pode ser o seguinte: realize um percurso na árvore, digamos pré-ordem; para os nodos

internos visitados codifique o *bit* 1, para os nodos folhas visitados codifique o *bit* 0.

Uma codificação fixa de prefixo pode ser utilizada para representar os símbolos do alfabeto. Cada símbolo será representado por $\log|S|$ *bits*.

Ao todo, a árvore de prefixo minimal pode ser representada com:

$$(2|S|-1)+|S|\log|S| \textit{ bits}.$$

Para uma ilustração desta forma de representação, fornecemos abaixo a codificação para a árvore da figura 2.1.

$$1\,1\,0\,0\,1\,1\,0\,0\,0\,c\,d\,b\,e\,a$$

Considere a árvore da figura 2.1. Sua altura tem tamanho 3, significando que o comprimento do maior código é 3. Mais importante do que a altura da árvore é o conceito de altura ponderada que passamos a introduzir.

Vamos associar a cada folha da árvore um *peso*, que corresponde ao quociente do número de ocorrências do símbolo s no texto fonte pela quantidade total de símbolos ocorrendo no mesmo texto. Ou seja, o peso de s é a freqüência relativa de s no texto fonte.

Definição 2.12 [Altura Ponderada] Seja p_s o peso associado ao símbolo s e l_s o seu nível na árvore. A *altura ponderada* de uma árvore de prefixo é expressa por:

$$h_p := \sum_{s \in S} p_s l_s \tag{2.1}$$

Pode-se notar que h_p é o comprimento médio do código para um símbolo fonte.

Observe a figura 2.3, que apresenta uma árvore semelhante à da figura 2.1, somente que com o acréscimo dos pesos internamente aos nodos, calculados com base na ocorrência dos símbolos no texto fonte. Ela constitui uma *árvore ponderada* e tem a característica de que seus nodos folhas têm pesos associados e o peso dos nodos internos é a soma dos pesos dos nodos filhos. Sua altura ponderada é de 2.2 *bits*, sendo também o comprimento médio para o código.

Se para os pesos dos nodos da árvore, for adotada, em substituição a freqüência relativa do símbolo, a freqüência f_s, que é o número de ocorrências de s no texto fonte, obtém-se a expressão:

$$\sum_{s \in S} f_s l_s$$

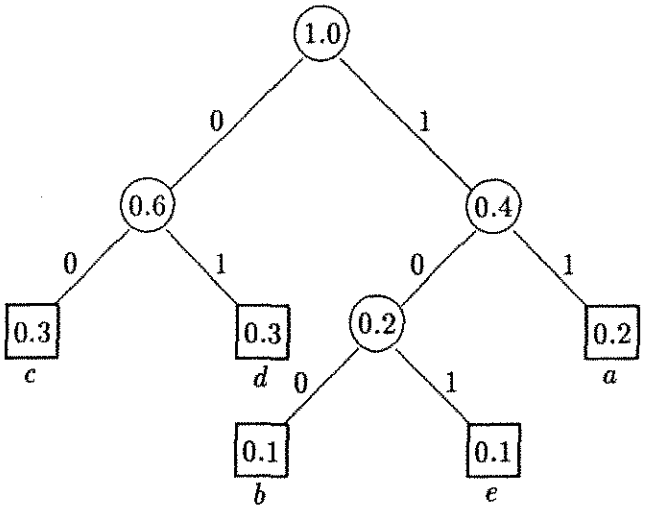


Figura 2.3: Árvore ponderada para o exemplo 2.9.

que equivale a:

$$h_p \sum_{s \in S} f_s$$

que vem a ser o comprimento total para o texto comprimido.

A árvore da figura 2.4 contém freqüências internamente aos nodos que correspondem às do exemplo 2.9. O comprimento total para o texto comprimido será de 22 *bits*.

2.2 Por Que a Compressão é Possível

Foi mencionado que o processo de compressão transforma o texto fonte num texto de comprimento menor que o original. Este processo de redução merece algumas explicações. Tem-se que não é possível reduzir sempre o tamanho de todos os textos. Um simples argumento de contagem vem a confirmar este fato. O que será possível é a diminuição do tamanho dos textos mais prováveis, às custas de se aumentar o tamanho dos textos menos prováveis. Iremos, nas próximas seções, esclarecer como é possível se efetuar reduções no comprimento dos textos.

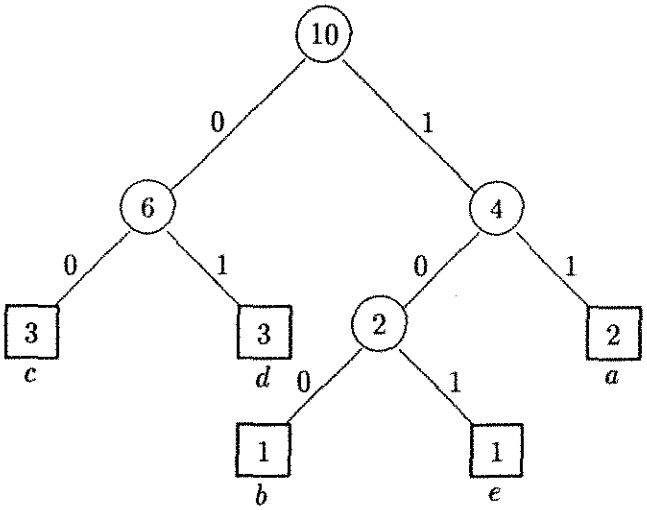


Figura 2.4: Árvore com freqüências para o exemplo 2.9.

2.2.1 Conceito Empírico de Redundância

A compressão que é possível efetuar sobre textos é devida à *redundância de informações* que esses apresentam. Esta redundância baseia-se em vários aspectos. De certa forma, há uma interpenetração entre estes aspectos; entretanto, em momentos específicos no texto, será possível distinguir uma tonalidade mais forte de um deles.

Inicialmente, temos o aspecto da *ocorrência de símbolos.* Quando representamos textos em códigos fixos como o ASCII, EBCDIC, só estamos preocupados em cobrir todas as possibilidades de ocorrências de símbolos pertencentes ao alfabeto ao qual o texto se fundamenta. No caso EBCDIC, como possuímos 256 símbolos no alfabeto, iremos precisar de $\log_2 256 = 8$ dígitos binários para representar cada símbolo. Possivelmente, nem todos os 256 símbolos estarão presentes no exemplo de texto de que estamos tratando; por exemplo, se for um texto em língua portuguesa, somando-se as letras maiúsculas e minúsculas, com acentos, caracteres de pontuação e algarismos temos cerca de 100 caracteres utilizados. Poderíamos pensar então em representar os caracteres de textos em português com apenas 7 *bits.*

Considerando um alfabeto formado somente por símbolos que ocorrem no texto, podemos aproveitar a freqüência que estes símbolos apresentam. Muitos desses símbolos aparecem com maior freqüência que outros, como a letra *a* nos

textos em português, outros com menor, como a letra *w*. Podemos pensar em representar o *a* com um número menor de *bits* e o *w* com um número maior. Se utilizarmos o mesmo raciocínio para as palavras, ou cadeias de símbolos, podemos reduzir ainda mais o tamanho do texto. Certas construções de línguas naturais são muito utilizadas, tais como as preposições, pronomes, e outras construções que ficam a sabor do escritor. Há certas palavras que se repetem muito em um texto específico como, no caso deste, a palavra "compressão"; há ainda cadeias de caracteres que sempre ocorrem juntas, com uma maior ou menor freqüência, como as seqüências *qu, ção, ch*, em textos em português e as seqüências de consoantes que dificilmente ocorrem. Em linguagens formais, destacamos as palavras reservadas e os nomes de identificadores. Enfim, esta *existência de padrões*, que se repetem em textos, é mais outro aspecto a ressaltar a redundância.

Merecem especial interesse alguns padrões que compõem cadeias formadas pela repetição do mesmo símbolo. É muito comum a ocorrência de cadeias de zeros ou brancos em textos de aplicações comerciais, ou cadeias de *pixels* de mesmo valor, freqüentes em textos de imagens sintéticas. Esta *repetição de símbolos* é outro aspecto que possibilita a redundância.

Há certos padrões que só ocorrem em determinados momentos no texto, e depois caem em desuso por longo período. É o que ocorreu, por ex., com a palavra "código", presente com muita freqüência na primeira parte deste Capítulo, e o que está ocorrendo com a palavra "redundância", no presente momento. Há exemplos mais apropriados, como certos padrões que tendem a se formar em textos de imagens. Este fato compõe uma redundância local a que chamamos de *localidade de referência.*

Em muitos dos aspectos de redundância podemos então associar um código, não aos caracteres individualmente, mas à sua combinação, e quanto maior sua freqüência, menor deverá ser o seu código.

É essencialmente com base na redundância de informações apresentada pelos textos que a compressão pode ser realizada. Esta redundância está presente a partir destes principais aspectos destacados de freqüência de símbolos, símbolos repetidos, existência de padrões e localidade de referência. Todos estes aspectos são decorrentes de uma certa estrutura do texto, ou da organização dos seus símbolos, o que justifica a sua interpenetração. Assim, quanto mais organizado estiver o texto, mais redundância este irá proporcionar, e portanto maior compressão será possível. Vale ressaltar que esta redundância é destacada somente com base na estrutura que o texto apresenta, não se levando em consideração nenhuma interpretação que se possa atribuir a cadeias de símbolos no texto fonte.

Os métodos que efetuam compressão sobre textos devem, portanto, encontrar e remover sua redundância. Esta, por sua vez, é peculiar a cada tipo de texto, e,

como os métodos pretendem ser universais, não se utilizando de nenhum recurso semântico para efetuar a compressão, estes devem "apreender" as características, ou a estrutura, que os textos apresentam. É justamente com base na aprendizagem das características dos textos que métodos são criados e sua eficiência analisada.

É interessante observar que a eliminação da redundância favorece a segurança de dados e conseqüentemente a Criptografia. Por outro lado, uma vez que a redundância é de grande utilidade para o tratamento de erros, sua remoção dificulta a detecção e correção dos mesmos. Alguns trabalhos recentes têm explorado a interação entre Compressão e Criptografia [WC88]. Lelewer e Hirschberg em [LH87] apresentam uma pequena discussão envolvendo a susceptibilidade a erros inerente a alguns métodos.

2.2.2 Conceito Formal de Redundância

O conceito que acabamos de introduzir apresenta a redundância como decorrente da forma como o texto se estrutura, ou como os símbolos fonte se distribuem no texto. Isto exige uma definição mais formal de redundância, que passamos a apresentar.

Fontes de Dados

Os textos são gerados por fontes de dados. Assim, existem fontes de línguas naturais, de linguagens formais, de dados numéricos, de imagens, e muitas outras.

Definição 2.13 [Fonte de Dados] Considere S o alfabeto de símbolos fonte. Uma *fonte de dados* é uma fonte que gera símbolos de S, e possui características estatísticas bem definidas. Ou seja, os símbolos em S têm probabilidades perfeitamente identificadas, cuja soma tem o valor 1.

Medida de Informação

A compressão, no seu processo, precisa remover a redundância presente no texto fonte, porém deve manter a informação que o texto apresenta. Entretanto, qual a quantidade de informação que um texto possui? Como ela se expressa em termos de símbolos binários?

Definição 2.14 [Medida de Informação] A *medida de informação* para um símbolo s, que possui probabilidade p_s é dada por:

$$\log_2 \frac{1}{p_s}$$

símbolo	probabilidade	código
the	1/2	00
and	1/4	01
her	1/8	10
for	1/8	11

Tabela 2.1: Tabela com alguns símbolos da língua inglesa.

A base da função log é dois porque a informação está medida em símbolos binários. No restante deste trabalho, quando não explícito, estaremos supondo que a função log tem base 2.

Se um símbolo é redundante, é como se ele não guardasse informação e fosse passível de redução. Viu-se que quanto mais freqüente é o símbolo, mais redundante será. Intuitivamente, se $p_s = 1$, temos que sua medida de informação é nula. Isto se explica, pois uma vez que o símbolo s sempre ocorre, ele não guarda nenhuma informação, é altamente previsível, e podemos representá-lo sem necessidade de código. Se p_s tem um valor médio, significa que s precisa ocorrer com outros símbolos para que possa representar alguma informação; logo seu código não deverá ser muito grande. Por outro lado, se p_s é bastante pequena, significa que s, raramente ocorrendo, guarda bastante informação, não é de forma alguma previsível, e teremos que representá-lo com um código de grande número de *bits*. Quanto mais redundante, mais previsível é o símbolo e menos informação este conterá.

Considerando-se a medida de informação para cada símbolo da fonte, pode-se encontrar uma média de informação para a fonte de dados.

Definição 2.15 [Entropia] A medida de informação de uma fonte de dados, ou *entropia da fonte*, ou *entropia para um símbolo da fonte* é expressa por:

$$H(fonte) = \sum_{s \in S} p_s \log \frac{1}{p_s} \tag{2.2}$$

Exemplo 2.16 Considere uma fonte de língua inglesa que gera os símbolos da tabela 2.1.

A entropia para a fonte, calculada com base na probabilidade dos símbolos, é de $H = 1.75$ *bits*/símbolo. Este valor expressa a quantidade média de *bits* para representar cada símbolo da fonte.

A entropia é uma medida do conteúdo da informação que uma fonte de dados apresenta. Como um texto compresso deve preservar a quantidade de informação do texto fonte, ela estima a menor quantidade de *bits*/símbolos necessária para guardar o conteúdo de informação da fonte, e portanto para representar textos recuperáveis emitidos por esta. Assim, a *entropia é considerada um limite para a compressão*, e pode ser tomada como uma medida de eficiência para os métodos de compressão.

Definição 2.17 [Comprimento Médio de Código] Seja S o alfabeto de símbolos fonte e C, o alfabeto de códigos. Considere l_s o comprimento do código para o símbolo s e p_s a sua probabilidade, o *comprimento médio de código* é expresso por:

$$C_m = \sum_{s \in S} p_s l_s \tag{2.3}$$

Observe que se o código é de prefixo, este valor expressa a altura ponderada, h_p, da árvore de prefixo.

Definição 2.18 [Redundância] Define-se *redundância* como sendo a diferença entre o comprimento médio de código e o valor da entropia. Assim:

$$R = C_m - H \tag{2.4}$$

Se a redundância se aproxima de zero — é um valor que está entre 0 e 1 — é porque o comprimento médio do código se aproxima do mínimo teórico e neste caso diz-se que o código tem *redundância mínima*. Em outras palavras:

$$H \leq C_m < H + 1 \tag{2.5}$$

Exemplo 2.19 Considere o código apresentado na tabela 2.1. O valor do C_m é de 2.0, e a redundância existente é de 0.25.

Todos estes conceitos abordados de medida de informação, entropia e redundância, juntamente com as suas conseqüências, foram apresentados primeiramente por Shannon em seu clássico trabalho de Teoria da Informação [Sha48]. Em particular, a relação 2.5, também conhecida como o *Primeiro Teorema de Shannon*, é seu mais importante resultado.

Observa-se que não tem sentido estimar um valor de entropia sem que as características estatísticas (símbolos com probabilidades) da fonte estejam perfeitamente identificadas. Ou seja, sem que se estabeleça um *modelo de fonte de dados*. Shannon estimou valores para fontes com características estatísticas bem definidas. Iremos em seguida qualificar algumas fontes.

2.2.3 Modelos de Fontes de Dados

Fontes Independentes

Nas fontes de dados independentes, a probabilidade associada a um símbolo independe dos outros símbolos. Neste caso, o valor até aqui apresentado de entropia não se modifica.

Um importante resultado é que a entropia é máxima quando os símbolos são gerados independentemente e de maneira uniforme. Sendo assim, não há como se efetuar compressão sobre o texto.

Dificilmente, na prática, os símbolos nos textos ocorrem independentemente. Os aspectos mencionados de redundância — existência de padrões, repetição de símbolos e mais fortemente localidade de referência — ratificam este fato.

Fontes Markovianas

Quando a geração de um símbolo depende da geração dos símbolos anteriores, diz-se que a fonte de dados é do tipo markoviana. Mais precisamente, uma *fonte markoviana de n-ésima ordem* é aquela em que a ocorrência do símbolo s depende da ocorrência dos n símbolos anteriores.[Abr63] Em particular, fontes independentes são ditas *fontes markovianas de ordem* 0, ou *fontes sem memória.*

No cálculo da entropia para fontes markovianas as probabilidades associadas aos símbolos devem ser substituídas por *probabilidades condicionais*, o que fornece uma nova fórmula de entropia. Decidimos adotar por simplicidade a fórmula genérica expressa por H (2.2).

Sabe-se, que a partir da adoção de um modelo de fonte markoviana mais complexo, que expresse uma real distribuição de probabilidade para os símbolos, tem-se o valor da entropia diminuído. Este aspecto estabelece que quanto mais um método considera, na sua abordagem, um inter-relacionamento entre os símbolos do alfabeto fonte, consegue melhores resultados de compressão.

Para informações mais detalhadas sobre os assuntos apresentados nestas duas últimas seções, i.é, seções 2.1 e 2.2, Abramson [Abr63] e Hamming [Ham80] fornecem excelentes estudos sobre Codificação e Teoria da Informação.

Na próxima seção, iremos propor uma classificação para os métodos de compressão, de forma a explorar os conceitos até aqui exibidos, e de maneira a expressar como os métodos confrontam-se com a modelagem dos dados.

2.3 Classificação para os Métodos de Compressão

Diante do que vimos nas seções anteriores, pode-se subdividir o processo de compressão em duas etapas:

1. **Modelagem dos dados** - Onde são determinadas as características estatísticas para o texto fonte.

2. **Codificação dos dados** - Onde é escolhida uma codificação apropriada, levando-se em consideração as características da fonte de dados.

Na apresentação de seu inovativo método de compressão, Lempel e Ziv [ZL77] estabeleceram claramente a distinção entre o seu trabalho e os demais métodos até então conhecidos. Até aquela época, existiam muitos métodos que propunham códigos de compressão com base em um conhecimento prévio das características estatísticas do texto fonte. Neste caso, o desafio estaria em encontrar códigos de redundância mínima. O novo método proposto, a partir de uma heurística que visava a aprendizagem das características do texto, exibia um processo conjunto de modelagem e de codificação. Com base neste reconhecimento, estabelecemos a seguinte distinção entre os métodos de compressão. A ilustração desta classificação será efetuada no restante deste trabalho, a partir da apresentação de diversos métodos de compressão.

- **Métodos Estatísticos**
  - Métodos que consideram a fonte de dados bem caracterizada;
  - Os códigos são associados aos símbolos com base nas suas probabilidades;
  - Permitem uma separação entre a modelagem e a codificação dos dados.
- **Métodos de Codificação por Fatores**
  - Métodos que não consideram a fonte de dados caracterizada;
  - Os códigos são associados a longas cadeias de símbolos consecutivos (fatores);
  - Exibem um processo conjunto de modelagem e codificação dos dados.

Nem sempre o reconhecimento das duas etapas, de modelagem e codificação, no processo de compressão, existiu. Da mesma forma, nem sempre a classificação entre os métodos, a partir da maneira como incorporam estas duas etapas, existiu. Este fato ocorria porque, em verdade, a distinção entre as duas etapas e

conseqüentemente entre duas categorias de métodos não é tão clara. Um fator complicador, porém muito mais interessante, é que recentemente demonstrou-se que qualquer método de codificação por fatores pode ser substituído por um método estatístico equivalente. [Lan83] [Bel87]

Bell em [BWC89] observa que somente nesta última década, principalmente a partir do trabalho de Rissanen e Langdon [RL81], a distinção entre as duas etapas da compressão tem sido objeto de preocupação. Esta preocupação deve-se à consideração de que o uso de métodos estatísticos, que combinam poderosos modeladores de dados com técnicas de codificação de redundância mínima, tem sido apontado como o caminho que poderá trazer resultados futuros para a compressão.

A distinção aqui proposta para os métodos acompanha as classificações mencionadas por Storer e Szymanski [SS82], Fraenkel *et alii* [FMP83], Crochemore [Cro89], Bell *et alii* [BWC89], Fiala e Greene [FG89] e Zipstein [Zip90].

Na próxima seção iremos apresentar tipos de modelagens possíveis para os dados.

2.3.1 Tipos de Modelagens de Dados

Modelagem Estática

Numa modelagem estática as probabilidades para os símbolos do alfabeto fonte são fixas. Isto significa que a associação entre símbolos e códigos é fixa, não mudando durante a compressão. Neste caso, tanto o compressor quanto o descompressor devem ter conhecimento do modelo de dados, ou da tabela de códigos, antes de iniciada a codificação propriamente dita.

Uma alternativa para a determinação das probabilidades é o método trabalhar com tabelas, que contém probabilidades previamente estabelecidas para certos tipos de dados. Assim, existiriam tabelas de língua portuguesa, de língua inglesa, de diversos códigos fonte, etc. Outra alternativa é o cálculo da freqüência de cada símbolo ocorrendo no texto fonte. Uma vez que as probabilidades são proporcionais às freqüências, a construção do código pode ser feita diretamente a partir das freqüências.[2]

McIntyre e Pechura em [MP85] apresentam uma comparação entre as duas alternativas na compressão de textos de códigos fonte em PASCAL, FORTRAN, COBOL e PL/I, utilizando-se do método de Huffman [Huf52], e salientam o bom

[2]Alguns autores [BWC89] designam estas duas alternativas mencionadas por *modelagem estática* e *modelagem semiadaptativa*, respectivamente.

desempenho da primeira. A segunda alternativa, entretanto, é a freqüentemente utilizada. Em parte pela dificuldade no estabelecimento das tabelas da primeira alternativa e principalmente porque o seu uso não expressa exatamente a distribuição dos símbolos no texto fonte. A segunda alternativa, contudo, tem o inconveniente de exigir duas leituras sobre o texto fonte, uma para o cálculo das freqüências e outra para a codificação do texto. Estas duas leituras não só aumentam o tempo de processamento requerido, mas tornam o algoritmo *off-line*. Além disso, como o descompressor deve ter conhecimento da tabela de códigos antes de iniciado o processo, e sendo ela específica para o texto fonte, deverá fazer parte do texto comprimido.

Esta modelagem é bastante útil a aplicações em que a associação de códigos não pode variar. Como exemplo, temos as bases de dados comprimidas [KBD89].

Modelagem Adaptativa

A modelagem adaptativa, também chamada de *modelagem dinâmica*, com base em uma única leitura sobre o texto fonte, considera as probabilidades até então calculadas para os símbolos ocorrendo no texto. Assim, os métodos com este funcionamento devem reconstruir dinamicamente a tabela de códigos de maneira a acompanhar as mudanças de probabilidades apresentadas pelos símbolos. Destarte, na compressão, a modelagem é realizada concomitantemente à codificação. Isto significa que tanto o alfabeto fonte como o alfabeto de códigos se modificam durante a compressão.

Como acontece na modelagem estática, a maioria dos métodos estima as probabilidades diretamente a partir das freqüências dos símbolos ocorrendo no texto fonte, ou das freqüências relativas. Com base neste fato, estaremos utilizando continuamente, no restante deste trabalho, o termo *peso* em substituição à probabilidade do símbolo.

Nesta modelagem, é preciso que o compressor e o descompressor, a partir de uma tabela inicial e com base no mesmo algoritmo, atualizem o modelo dinamicamente, mantendo uma sincronia. A tabela inicial, adotada por alguns métodos, supõe uma distribuição equiprovável para os símbolos do alfabeto. Alguns outros métodos iniciam com uma tabela sem nenhum símbolo; à medida que novos símbolos vão ocorrendo no texto, a sua presença no modelo é indicada para o descompressor através de uma codificação apropriada. A atualização do modelo é feita da seguinte maneira: quando um símbolo é lido, o compressor primeiramente o codifica e só depois atualiza o seu peso. O descompressor, ao ler o código, reconhece o símbolo e, da mesma forma que o compressor, atualiza o seu peso. É importante verificar que para a descompressão ser possível, o código gerado pelo

compressor para o próximo símbolo é sempre relativo à tabela de códigos anterior à atualização do seu peso. Da mesma forma, o código para um novo símbolo não previsto no modelo deve ser de conhecimento do descompressor.

O processo descrito acima, além de evitar duas leituras sobre o texto fonte, não exige que o modelo de dados seja enviado no texto comprimido. Isto permite que os métodos com esta modelagem apresentem um algoritmo *on-line*, significando que na compressão de um texto, após a codificação de um símbolo, este já pode ser decodificado. Ou seja, a descompressão pode ser realizada concomitantemente à compressão.

A modelagem adaptativa é útil para aplicações *on-line*, tal como a comunicação de dados, mas a sua principal importância é que os métodos que a adotam têm um desempenho na prática igual e às vezes superior do que se adotassem uma modelagem estática [BWC89]. Sendo assim, os melhores métodos de compressão que iremos apresentar exibem um funcionamento adaptativo.

Como pode ser observado nesta última seção, estamos utilizando indistintamente os termos "modelagem" e "funcionamento" para os métodos. Esta confusão natural será perpetuada no restante deste trabalho.

Capítulo 3

Métodos de Compressão

Este Capítulo apresenta nas duas primeiras seções os métodos de compressão estatísticos e os métodos de codificação por fatores. Para cada método será feita uma *descrição* da maneira como constroem seu código e como modelam seus dados. Será também apresentada uma *avaliação*, principalmente a nível teórico, considerando o desempenho de compressão alcançado pelo método. Implementações existentes serão descritas e, em particular, a *implementação* utilizada para a realização dos experimentos do último Capítulo será abordada em detalhes. A última seção apresenta compressores genéricos, alguns comerciais, outros de livre distribuição. Na sua descrição, procurou-se, na medida do possível, associar as várias modalidades de compressão exibidas com os métodos apresentados nas duas seções anteriores.

3.1 Métodos Estatísticos

Estes métodos apresentam uma codificação que depende de uma modelagem dos dados, ou seja, de uma caracterização prévia da fonte de dados. Isto significa que, com base no conhecimento das probabilidades dos símbolos do alfabeto fonte considerado, eles apresentam uma codificação que geralmente é de redundância mínima. Desta forma, seu objetivo é associar códigos maiores a símbolos menos prováveis e códigos menores a símbolos mais prováveis.

A modelagem que estes métodos costumam adotar freqüentemente supõe uma fonte de dados como sendo de ordem 0, ou seja, uma fonte independente. Neste caso, escolhem um alfabeto fonte, geralmente formado por caracteres de representação da máquina (ASCII, EBCDIC, etc.), ou de palavras bem-definidas, tais como as palavras reservadas de uma linguagem de programação, e tanto podem modelar os símbolos de forma estática como adaptativa.

Embora estes métodos estatísticos apresentem redundância mínima, na prática seu desempenho está associado ao tipo de modelo de dados que está sendo adotado. Se eles adotam uma modelagem simples, como a independente, seu desempenho é muito fraco, comparativamente, por ex., aos métodos de codificação por fatores. Um outro fator que diferencia estes métodos é que nem sempre a codificação utilizada por eles consegue se adequar facilmente e eficientemente a várias formas de modelagem.

Somente recentemente, alguns autores têm se preocupado exclusivamente com uma modelagem adaptativa de mais alta ordem, e têm apresentado métodos eficientes que a realizam. Abrahamson [Abr89] propõe um modelo que explora a interdependência entre caracteres sucessivos; Cormack e Horspool [CH87] e Llewellyn [Lle87] exibem modelos mais robustos, baseados em máquinas de estados finitos. Estes modeladores têm a principal desvantagem de requererem muita memória interna; entretanto, atuando principalmente com a codificação aritmética [WNC87], têm se constituído em poderosos compressores.

Os métodos estatísticos a serem apresentados nesta seção são o método de Shannon-Fano [SW49] [Fan49], o método de Huffman [Huf52] [Fal73] [Gal78] [Knu85] e o método Aritmético [Abr63] [Lan84] [WNC87]. Adiantamos que a sua apresentação descreverá principalmente a codificação proposta. As modelagens mais freqüentes serão mencionadas, entretanto formas de modelagem de mais alta ordem não serão detalhadas. Para maiores informações sobre formas de modelagem, Bell *et alii* em [BWC89] fornecem um estudo bastante minucioso.

3.1.1 Método de Shannon-Fano (1949)

Este método foi criado independentemente por Shannon [SW49] e por Fano [Fan49] com o objetivo de se obter um código cujo comprimento médio se aproximasse da entropia, ou como foi definido, código de redundância mínima. Este intento, entretanto, não foi satisfatoriamente alcançado, o que levou Huffman [Huf52], posteriormente, à criação de seu conhecido código de redundância mínima, onde finalmente, consegue o pretendido.

Descrições do método Shannon-Fano foram encontradas em [SW49], [LH87], [Hel87] e [Sto88].

Descrição

Dado um alfabeto fonte, cujos símbolos têm pesos determinados, o método apresenta um código de prefixo minimal. O objetivo será associar códigos menores a símbolos mais prováveis e códigos maiores a símbolos menos prováveis. O código tem comprimento necessariamente variável, enquanto que os símbolos podem apresentar comprimento fixo ou variável.

Código de Shannon-Fano O código é obtido a partir da construção de uma árvore ponderada de prefixo de maneira *top-down*, que tem por base os pesos existentes para os símbolos.

A construção da árvore é a seguinte:

1. Enumera-se os símbolos fonte na ordem não-crescente dos seus pesos. Associa-se esta lista ao nodo raiz da árvore a ser construída;

2. Subdivide-se a lista enumerada em duas sublistas, sem alterar a ordem dos símbolos. A diferença entre as somas dos pesos de cada sublista deve ser mínima. Isto significa que uma sublista deve ter um peso total aproximadamente igual ao da outra;

3. As duas sublistas comporão nodos filhos do nodo correspondente à lista que lhes deu origem;

4. Repete-se este processo de subdivisão até que as sublistas geradas sejam unitárias.

Na figura 3.1, fornecemos um algoritmo para a construção da árvore que esclarece melhor o processo. Em verdade, esta descrição construtiva do código é devida a Fano [Fan49]. Shannon em [SW49] apresenta uma descrição aritmética para o

1. Seja L a lista de símbolos do alfabeto fonte, e P a lista dos respectivos pesos dos símbolos em ordem não-crescente.
   $L := \{s_1, s_2, \ldots, s_n\}, \quad P := \{p_1, p_2, \ldots, p_n\}$.

2. Considere n_r o nodo raiz da árvore de prefixo.
   Associe L à raiz da árvore: (L, n_r).

3. Para (L, n_r), com L não unitária, faça

   (a) Divida L em L_e e L_d,
   $L_e := \{s_1, s_2, \ldots, s_k\}$ e $L_d := \{s_{k+1}, s_2, \ldots, s_n\}$, $p_1 \geq p_2 \geq \ldots \geq p_n$,
   $|\sum_{i=1}^{k} p_i - \sum_{i=k+1}^{n} p_i|$ é mínimo, $1 \leq k < n$.

   (b) Considere n_e e n_d nodos filhos de n_r.
   Associe L_e a n_e: (L_e, n_e) e L_d a n_d: (L_d, n_d).
   Associe o bit '0' à aresta de n_r a n_e.
   Associe o bit '1' à aresta de n_r a n_d.

   (c) Repita o passo de número 3, substituindo (L, n_r) por (L_e, n_e) e por (L_d, n_d).

Figura 3.1: **Algoritmo para a construção da árvore de Shannon-Fano.**

código, visando com isso a obtenção de uma nova maneira para se demonstrar o seu *Primeiro Teorema* (2.5). Esta mesma descrição de Shannon principiou uma outra forma de codificação, que algum tempo depois seria conhecida como "codificação aritmética". Este método aritmético será abordado em detalhes.

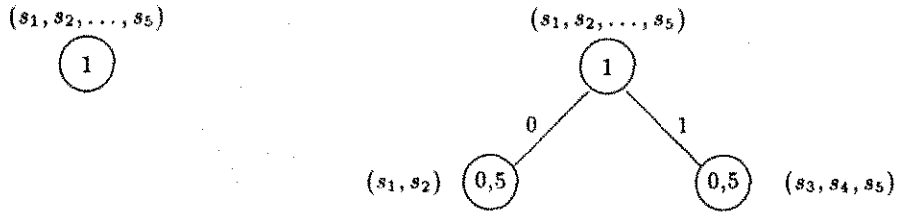
Vamos ilustrar a construção do código a partir do exemplo que segue:

Exemplo 3.1 Considere $L := \{s_1, s_2, s_3, s_4, s_5\}$, com respectivos pesos em ordem não-crescente $P := \{0,3;\ 0,2;\ 0,2;\ 0,2;\ 0,1\}$.

Inicialmente, a árvore de prefixo será expressa pela figura 3.2 (a). Subdividindo-se L, obtemos: $L_e := \{s_1, s_2\}$, $P_e := \{0,3;\ 0,2\}$ e $L_d := \{s_3, s_4, s_5\}$, $P_d := \{0,2;\ 0,2;\ 0,1\}$, e a árvore de prefixo será a expressa pela figura 3.2 (b). Ao final do processo, obtém-se a árvore da figura 3.3, que estabelece o código descrito na tabela 3.1.

A altura ponderada para a árvore será de:

$$h_p = 0,3 * 2 + 0,2 * 2 + 0,2 * 2 + 0,2 * 3 + 0,1 * 3 = 2,30 \; bits.$$



(a) Árvore inicial. (b) Árvore após primeiro passo.

Figura 3.2: Construção do código de Shannon-Fano para o exemplo 3.1.

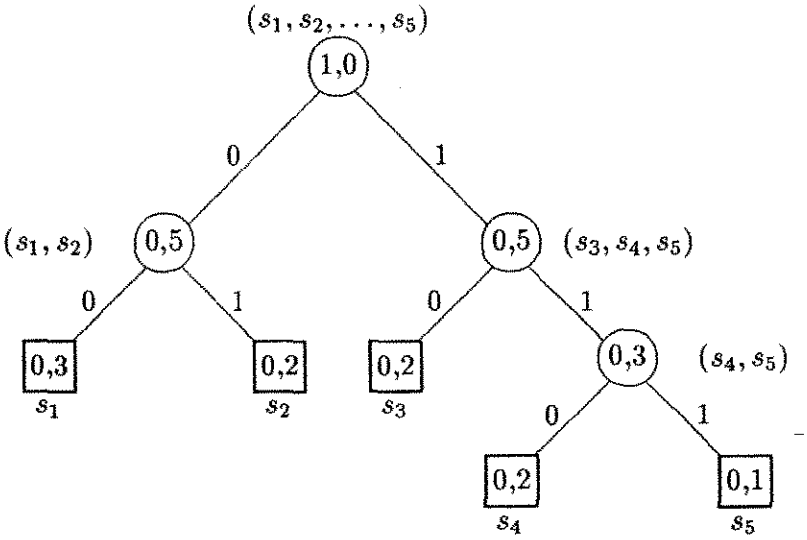


Figura 3.3: Árvore de Shannon-Fano para o exemplo 3.1.

símbolo	peso	código	comprimento
s_1	0,3	00	2
s_2	0,2	01	2
s_3	0,2	10	2
s_4	0,2	110	3
s_5	0,1	111	3

Tabela 3.1: Codificação de Shannon-Fano para o exemplo 3.1.

A entropia será de :

$$H = 0,3*\log\frac{10}{3} + 0,2*\log\frac{10}{2} + 0,2*\log\frac{10}{2} + 0,2*\log\frac{10}{2} + 0,1*\log\frac{10}{1} = 2,246\ bits.$$

A redundância será portanto de $0,054$ *bits*.

O método de Shannon-Fano tanto pode efetuar uma modelagem estática quanto dinâmica dos dados. Entretanto, não temos conhecimento de uma algoritmo eficiente para o método que funcione de forma adaptativa.

O exemplo que segue aplica uma modelagem estática para a compressão de um texto e considera os símbolos ocorrendo independentemente. Considerações sobre o uso do código de Shannon-Fano com modelagens de mais alta ordem, além da independente, serão realizadas na próxima seção (3.1.2), quando da apresentação dos códigos de Huffman [Huf52].

Exemplo 3.2 Suponha que desejamos comprimir o texto: *carro caro*. O alfabeto fonte será $S := \{o,\ c,\ a,\ r,\ \sqcup\}$, onde $\sqcup$ representa o espaço em branco entre as palavras.

Inicialmente, o texto deve ser lido para o cálculo das freqüências. As freqüências para os símbolos do alfabeto, respectivamente, serão $P := \{2, 2, 2, 3, 1\}$. Posteriormente, a árvore de Shannon-Fano deve ser construída conforme o algoritmo apresentado na figura 3.1. A árvore encontrada é a da figura 3.3, a mesma do exemplo 3.1, já que os símbolos ocorrem na mesma proporção. Associando os símbolos em L aos símbolos em S, temos: $s_1 = \mathrm{r}$, $s_2 = \mathrm{c}$, $s_3 = \mathrm{a}$, $s_4 = \mathrm{o}$, $s_5 = \sqcup$. Finalmente, os símbolos devem ser substituídos pelo código encontrado, dando origem ao seguinte texto compresso:

$$01\,10\,00\,00\,110\,111\,01\,10\,00\,110,$$

cujo comprimento é de 23 *bits*.

Se o texto fonte fosse codificado em ASCII, onde cada caractere é representado por 8 *bits*, seu comprimento seria de 80 *bits*. Isto nos leva a uma taxa de compressão de 71%. Entretanto, o texto comprimido também deverá conter a árvore construída. Neste caso, o seu comprimento aumenta, diminuindo a taxa de compressão. Em termos práticos, o tamanho da árvore, que é proporcional ao tamanho do alfabeto (conforme apresentado na seção 2.1.1, p. 15), é bem menor que o comprimento do texto, não aumentando significativamente o comprimento final do texto comprimido.

A seguir, apresentamos um exemplo que comprova a não-optimalidade dos códigos de Shannon-Fano. Fornecemos, juntamente com a árvore de Shannon-Fano construída, uma árvore ótima e que foi construída a partir do método de Huffman [Huf52], a ser apresentado na próxima seção (3.1.2).

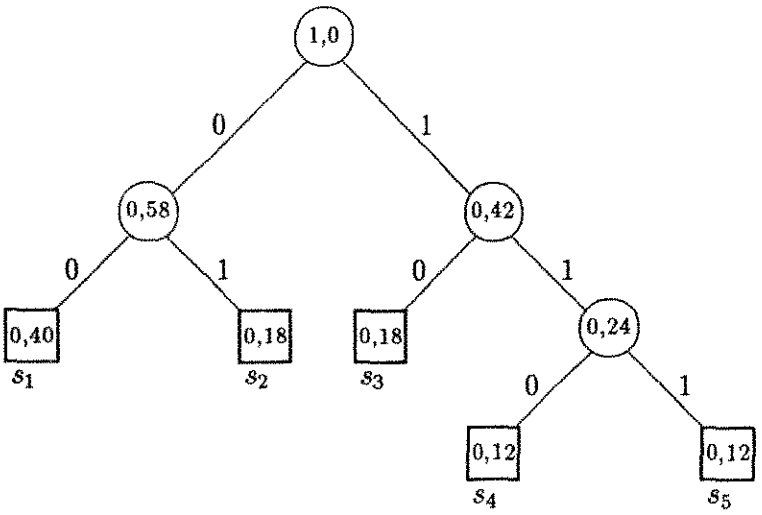


Figura 3.4: Árvore de Shannon-Fano para o exemplo 3.3.

Exemplo 3.3 Seja $L := \{s_1, s_2, s_3, s_4, s_5\}$ a lista de símbolos cujos respectivos pesos em ordem não-crescente são $P := \{0,40;\ 0,18;\ 0,18;\ 0,12;\ 0,12\}$.

As árvores de Shannon-Fano e de Huffman estão expressas pelas figuras 3.4 e 3.5, respectivamente. Enquanto a altura ponderada para a árvore de Huffman é de $2,20$ *bits*, a da árvore de Shannon-Fano é de $2,24$ *bits*.

Avaliação

A construção do código de Shannon-Fano não é *ótima*, significando que não há a garantia de se obter a menor altura ponderada possível, para um conjunto de símbolos com pesos bem definidos. O exemplo 3.3 ilustra este fato. A codificação, entretanto, atinge a entropia à medida que o comprimento do alfabeto tende a infinito [Huf52]. Demonstra-se que a altura ponderada do código de Shannon-Fano está entre a entropia e a entropia mais um, [Ham80] ou seja:

$$H \leq h_p < H + 1.$$

A redundância só será totalmente eliminada se os pesos para os símbolos fonte são potências inteiras de $1/2$. Sendo assim, o comprimento do código para o símbolo s, que apresenta peso p_s, será exatamente a medida de informação, i.é, $\log \frac{1}{p_s}$. Observe que neste caso, o particionamento da lista de símbolos, na construção do código, será sempre feita de forma igualitária.

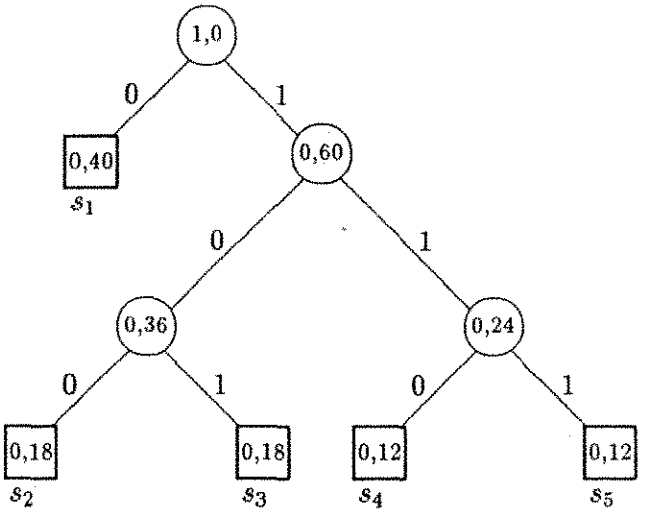


Figura 3.5: Árvore de Huffman para o exemplo 3.3.

Freqüentemente entretanto, a condição acima não se verifica. Neste caso, tomando-se l_s como o comprimento de código para o símbolo s, seu valor será o inteiro que obedecerá a relação abaixo:[Ham80]

$$\log \frac{1}{p_s} \leq l_s < \log \frac{1}{p_s} + 1. \tag{3.1}$$

A codificação proposta originalmente por Shannon [SW49] baseia-se no conhecimento do comprimento do código para cada símbolo, estabelecido pela relação acima (3.1). Nesta codificação, os símbolos inicialmente devem ser enumerados pela ordem não-crescente dos seus pesos. Para cada símbolo é associado um *peso cumulativo* P_s, que vem a ser o somatório dos pesos dos símbolos que o precedem na ordem estabelecida. Assim,

$$P_s = \sum_{i=1}^{s-1} p_i. \tag{3.2}$$

O código resultante para cada símbolo será a codificação binária do peso cumulativo e deverá ter comprimento l_s. A tabela 3.2 apresenta esta codificação original de Shannon para o exemplo 3.3.

Hamming em [Ham80], também explora este conhecimento do comprimento do código para cada símbolo, e propõe uma maneira simples para a sua construção. Inicialmente, os símbolos também devem ser enumerados na ordem não-crescente dos pesos. Posteriormente, códigos binários devem ser atribuídos a cada

símbolo	peso	peso cumulativo	comprimento	código
s_1	0,40	0,0	2	00
s_2	0,18	0,40	3	011
s_3	0,18	0,58	3	100
s_4	0,12	0,76	4	1100
s_5	0,12	0,88	4	1110

Tabela 3.2: Codificação original de Shannon para o exemplo 3.3.

símbolo	peso	comprimento	código
s_1	0,40	2	00
s_2	0,18	3	010
s_3	0,18	3	011
s_4	0,12	4	1000
s_5	0,12	4	1001

Tabela 3.3: Codificação de Shannon-Hamming para o exemplo 3.3.

símbolo, numa seqüência crescente, de forma a serem de prefixo e de maneira a obedecer o seu comprimento. Este processo foi o utilizado para a obtenção da tabela 3.3, que equivale a um código para o exemplo 3.3.

Embora as formas de codificação de Shannon e Hamming sejam eficientes, porque, em princípio, dependem apenas dos pesos individuais dos símbolos, e não de todo o conjunto, tal qual o método de Huffman [Huf52]; tem-se que altura ponderada dos dois códigos para o exemplo, que vem a ser de 2,82 *bits*, é bem inferior ao resultado obtido anteriormente. Ademais, uma observação mais atenta das tabelas 3.2 e 3.3, verifica que os códigos gerados são de prefixo, mas não minimais.

Implementação

A implementação utilizada nos experimentos do Capítulo 4 se baseou no algoritmo 3.1 para a construção da árvore. Neste algoritmo, o único fator complicador é o particionamento das listas, descrito no passo 3 (a). Entretanto, uma vez que é preciso manter a ordem estabelecida pelos símbolos, basta que se utilize a seguinte estratégia: percorre-se a lista até que se alcance um subpeso igual ou superior à metade do seu peso. Obtém-se então duas sublistas. A primeira, contendo os símbolos percorridos, de peso maior que a segunda. Para saber se este é o particionamento adequado, deve-se verificar se o peso da primeira sublista

se aproxima mais da metade da lista do que o peso desta mesma sublista sem seu último símbolo; em caso negativo, este último símbolo deverá passar para a segunda sublista. Considerando-se $|S|$ como o tamanho do alfabeto de entrada, a ordenação da lista, mencionada no passo 1 do algoritmo, tem complexidade $O(|S|\log|S|)$; pode-se provar que o particionamento total também tem esta complexidade. Portanto, a construção da árvore tem tempo $O(|S|\log|S|)$.

Por apresentar um código de prefixo, tanto a codificação quanto a decodificação do método são realizadas em tempo real. O processo utilizado é o mesmo descrito na seção 2.1.1, p. 14, que apresenta as árvores de prefixo.

Para a representação da árvore de prefixo no início do texto compresso utilizou-se o algoritmo definido na seção 2.1.1, p. 15.

Considerando-se N como o comprimento do texto fonte, o processo completo de compressão e descompressão é realizado em tempo $O(N)$. As estruturas de dados envolvidas ocupam espaço $O(|S|)$.

3.1.2 Método de Huffman (1952)

O método foi criado por D. A. Huffman [Huf52] com o objetivo de se obter os códigos de redundância mínima desejados por Shannon [SW49]. Huffman atinge este intento e obtém um código de prefixo *ótimo*, cuja árvore de prefixo tem a característica de apresentar a menor altura ponderada possível, a que ele chamou de *comprimento médio de mensagem*.

O método, que vem a ser um dos primeiros métodos universais a apresentar um resultado teórico *ótimo*, torna-se um padrão de comparação para outros métodos que posteriormente irão surgir.

Afora a sua importância para a compressão de dados, os códigos de Huffman ainda são empregados em muitas outras áreas. Lelewer e Hirschberg em [LH87] apresentam uma lista das suas muitas aplicações. Dentre elas podemos citar a determinação de um algoritmo para a determinação de árvores ótimas de busca.

Recentemente, com a redescoberta da codificação aritmética [WNC87] e o surgimento de formas de modelagem de dados de mais alta ordem, tornou-se clara a inadequação do uso do código de Huffman com este tipo de modelagem. Esta inadequação tem sido apontada como uma significativa desvantagem que desqualifica o uso do método na composição de compressores mais poderosos.

O método de Huffman para a construção de árvores ponderadas ótimas é um algoritmo clássico que resume várias aplicações, tendo sido apresentado em muitos textos. Estudos introdutórios, concernentes à compressão, foram encontrados em [LH87] e [Sto88].

Descrição

Considera-se a existência de um alfabeto fonte, cujos símbolos têm pesos determinados. O objetivo será a construção de um código de prefixo minimal que seja de redundância mínima. Sendo assim, tem por princípio associar códigos menores a símbolos mais prováveis e códigos maiores a símbolos menos prováveis. O código necessariamente terá comprimento variável, enquanto que os símbolos podem apresentar comprimento fixo ou variável.

Código de Huffman O código de Huffman é obtido a partir da construção de uma árvore de prefixo ponderada de maneira *bottom-up*.

A árvore de Huffman é assim obtida:

1. Considere uma floresta em que cada árvore tem sua raiz associada a um símbolo do alfabeto com seu respectivo peso;

2. Remova quaisquer duas árvores cujas raízes têm menor peso.
   Acrescente uma nova árvore que tem uma raiz cujos filhos são as árvores anteriores e cujo peso é a soma dos pesos das raízes dessas árvores;

3. Repita o passo anterior até que exista somente uma árvore na floresta.

Este procedimento estabelece a seguinte definição indutiva para os códigos de Huffman. Esta definição será utilizada na verificação de muitos dos seus aspectos.

Definição 3.4 [Árvore de Huffman] Uma árvore de Huffman é assim definida:

- Considere uma floresta formada por n árvores ponderadas, cujas raízes são representadas pelos seguintes pesos: $p_1, p_2, \ldots, p_n$.
- Substitua quaisquer árvores p_1 e p_2, de menores pesos, por $p_{1,2} = p_1 + p_2$.
- Construa a árvore de Huffman para as seguintes $n-1$ árvores da floresta: $p_{1,2}, p_3, \ldots, p_n$.
- Substitua na árvore construída o nodo $p_{1,2}$ pela subárvore cuja raiz é o próprio nodo e cujos filhos são p_1 e p_2.

Na figura 3.6, apresentamos um algoritmo para a construção da árvore que, para eficiência de implementação, se utiliza de uma lista de símbolos ordenada pelos pesos. Ao final do processo, terá sido construída uma árvore de prefixo cuja altura ponderada é a menor possível. O código obtido é portanto ótimo. A seguir, este fato será demonstrado.

Lema 3.5 *Considere uma árvore T de altura ponderada h_T. Sejam x e y duas folhas de T, com respectivos pesos p_x e p_y e alturas l_x e l_y. Se permutarmos as duas folhas, a nova árvore terá altura ponderada igual a $h_T + (l_x - l_y)(p_y - p_x)$.*

Demonstração. A altura da nova árvore é igual a:

$$h_T - l_x p_x - l_y p_y + l_x p_y + l_y p_x. \square$$

Teorema 3.6 (Código de Huffman é ótimo) *A árvore de Huffman apresenta a menor altura ponderada.*

Demonstração. Considerando uma árvore formada por n folhas, será feita a partir de uma indução sobre n.

Considere uma árvore T, que apresenta a menor altura ponderada h_T. Inicialmente, temos que T deve ser *cheia*, pois caso não fosse, os nodos que possuem apenas um filho poderiam ser eliminados.

Sejam p_1 e p_2 duas folhas de menor peso. Considere um nodo interno da árvore, digamos $p_{1,2}$, que apresenta a maior distância, digamos l, para a raiz. Se algum dos filhos de $p_{1,2}$ não é p_1 ou p_2, troque esse filho com p_1 ou p_2; o lema 3.5 garante que não há um aumento de h_T.

Considere uma árvore T', de altura ponderada $h_{T'}$, formada pela substituição de p_1 e p_2 em T por $p_{1,2}$, de acordo com a definição 3.4.

É fácil verificar que h_T se relaciona com $h_{T'}$. Pela definição 2.1 de altura ponderada, sabe-se que p_1 e p_2 contribuem com o seguinte valor para h_T: $(p_1 + p_2)\,l$. Para $h_{T'}$, $p_{1,2}$ contribui com: $(p_1 + p_2)\,(l-1)$. Portanto, tem-se que:

$$h_T = \sum_{i=3}^{n} p_i l_i + (p_1 + p_2)\,l \quad e \quad h_{T'} = \sum_{i=3}^{n} p_i l_i + (p_1 + p_2)\,(l-1).$$

E assim:

$$h_T = h_{T'} + p_1 + p_2.$$

Seja F uma árvore de Huffman com o mesmo conjunto de pesos de T. Seja F' a árvore de Huffman obtida a partir de F pela substituição de p_1 e p_2 por $p_{1,2}$. Por hipótese de indução, $h_{F'} \leq h_{T'}$. Analogamente à igualdade acima, temos que: $h_F = h_{F'} + p_1 + p_2$. Portanto, $h_F \leq h_T$. $\square$

Vamos ilustrar a construção do código a partir do exemplo seguinte.

Exemplo 3.7 Seja $L := \{s_1, s_2, s_3, s_4, s_5\}$ a lista de símbolos, cujos respectivos pesos, em ordem não-decrescente, são $P := \{0,1;\ 0,1;\ 0,2;\ 0,2;\ 0,4\}$.

1. Seja L a lista de símbolos do alfabeto fonte, e P a lista dos respectivos pesos dos símbolos em ordem não-decrescente.
   $L := \{s_1, s_2, \ldots, s_n\}, \quad P := \{p_1, p_2, \ldots, p_n\}$.
2. Enquanto L não é unitária faça
   (a) Remover s_1 e s_2, de menores pesos, de L.
   (b) Inserir na ordem requerida $s_{1,2}$ em L, com $p_{1,2} = p_1 + p_2$.
   (c) Criar nodo $s_{1,2}$ para a árvore, com filhos s_1 e s_2.
   Associar o *bit* '0' à aresta esquerda e o *bit* '1' à aresta direita.
3. Associar único símbolo em L à raiz da árvore.

Figura 3.6: **Algoritmo para a construção do código de Huffman.**

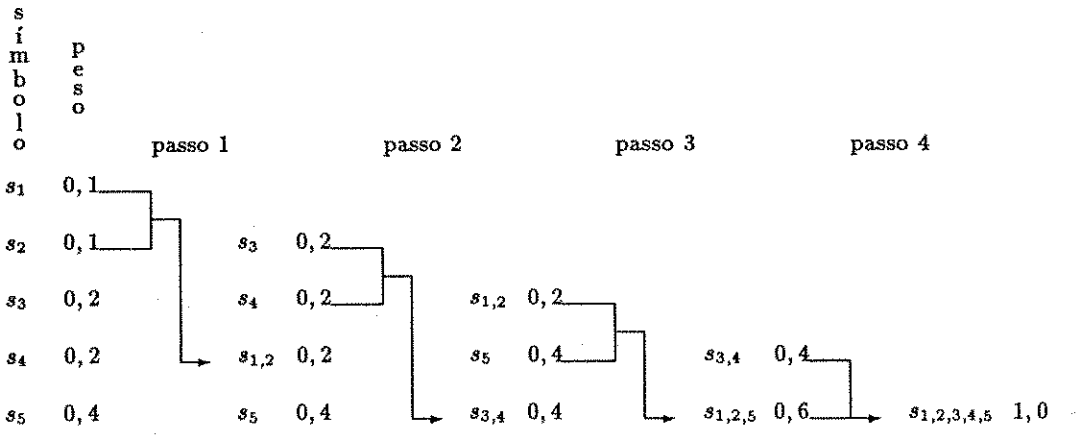
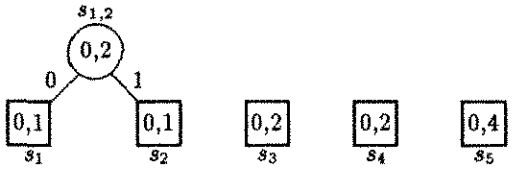


Figura 3.7: Atualização da lista de símbolos para o exemplo 3.7.

(a) Floresta inicial.



(b) Floresta após primeiro passo.

Figura 3.8: Construção inicial para a árvore de Huffman do exemplo 3.7.

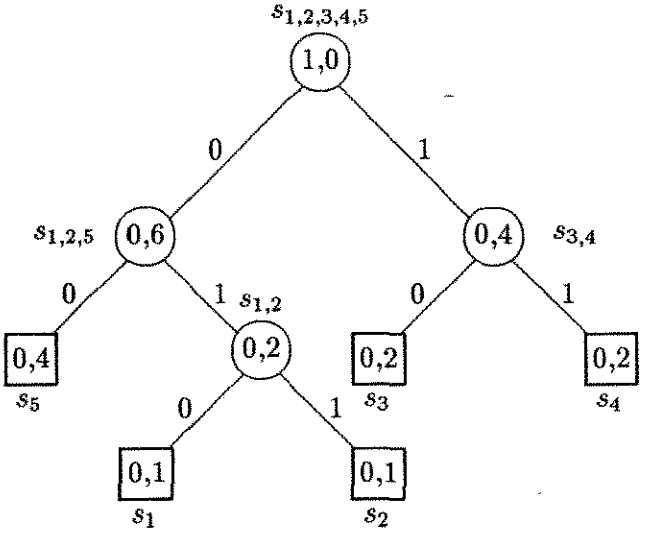


Figura 3.9: Árvore de Huffman para o exemplo 3.7.

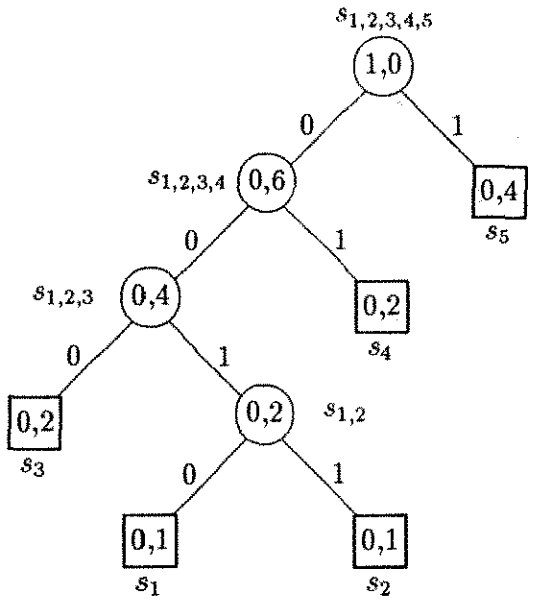


Figura 3.10: Outra árvore de Huffman para o exemplo 3.7.

O processo de atualização da lista L para a construção do código está expresso na figura 3.7. A floresta inicial será expressa pela figura 3.8. A árvore final de Huffman está expressa na figura 3.9. A altura ponderada para esta árvore será de:

$$h_p = 2(0,1 * 3) + 2(0,2 * 2) + 0,4 * 2 = 2,20 \; bits.$$

A entropia será de:

$$H = 2(0,1 * \log \frac{10}{1}) + 2(0,2 * \log \frac{10}{2}) + 0,4 * \log \frac{10}{4} = 2,12 \; bits.$$

A redundância existente será portanto de $0,08$ *bits*.

No algoritmo de Huffman, a escolha dos menores símbolos a serem removidos é arbitrária. Isto pode ser observado através do exemplo 3.7, quando nos passos 2 e 3 da figura 3.7 podemos optar entre vários símbolos a remover. Esta escolha não faz diferença, pois o que importa para o algoritmo são os pesos, independentemente dos símbolos associados. Assim, quaisquer que sejam os símbolos, a altura ponderada resultará sempre igual. Na figura 3.10, apresentamos outra árvore para o exemplo 3.7, construída através da escolha de outros símbolos, mas igualmente válida, conforme se verifica pela sua altura ponderada que também é:

$$h_p = 2(0,1 * 4) + 0,2 * 3 + 0,2 * 2 + 0,4 * 1 = 2,20 \; bits.$$

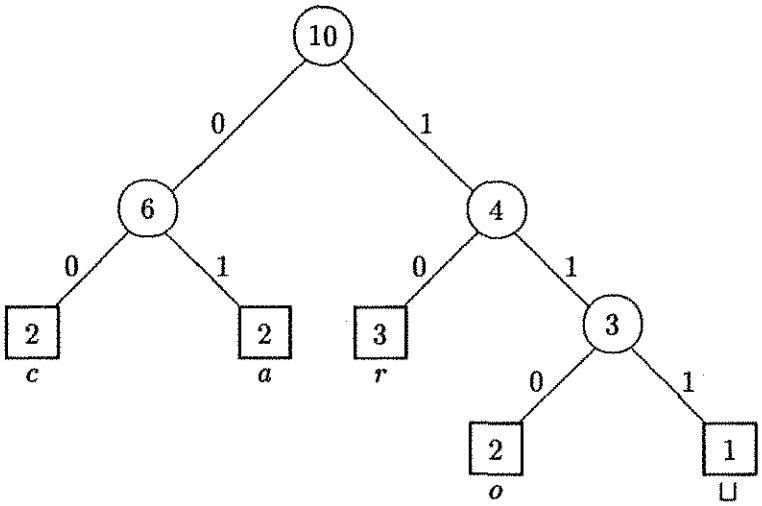


Figura 3.11: Árvore de Huffman para o exemplo 3.2.

A escolha dos símbolos pode ser interessante, conforme observa Schwartz em [Sch64], para diminuir o comprimento do maior código gerado, ou seja, da altura da árvore, assim como do caminho externo da árvore.[1] Neste caso, o novo símbolo a ser inserido na lista deverá ser colocado numa ordem maior que os demais de mesmo peso, sendo a última possibilidade de escolha na próxima remoção. Este critério foi o utilizado para a construção da árvore 3.9, que apresenta a menor altura e o menor caminho externo dentre as demais árvores que minimizam a altura ponderada. Hamming em [Ham80] observa que esta estratégia também permite a geração de um código de variância mínima.

O método de Huffman tanto pode modelar os dados de forma estática quanto de forma dinâmica. O método original, proposto em 1952, considerava uma modelagem estática para o texto. Posteriormente surgiram algoritmos eficientes para a construção dinâmica dos códigos de Huffman. Esses algoritmos serão apresentados na próxima seção (3.1.3).

A utilização do método de Huffman de maneira estática freqüentemente considera um alfabeto de caracteres ocorrendo independentemente no texto. Para uma ilustração deste uso, considere o exemplo 3.2, que também foi utilizado para ilustrar o método de Shannon-Fano. Após a determinação do alfabeto e o cálculo das freqüências para os símbolos, o código deve ser construído. A figura 3.11 apresenta a árvore construída para o exemplo. Omitindo-se a representação da

[1] Do inglês, *external path length*.

árvore, o texto original (*carro caro*) será codificado como:

$$00\,01\,10\,10\,110\,111\,00\,01\,10\,110$$

Observa-se que os dois métodos obtém a mesma altura ponderada, que é de 2,3 *bits*/símbolo. Entretanto, o exemplo 3.3, apresentado na seção anterior (3.1.1), ilustra a superioridade do código de Huffman em relação ao código de Shannon-Fano.

Avaliação

O método garante a obtenção de códigos de prefixo de redundância mínima, significando que se aproxima do mínimo teórico, que é a entropia. Entretanto, só eliminará toda a redundância se os pesos associados aos símbolos do alfabeto fonte são potências inteiras de 1/2. Neste caso, o comprimento do código — que deve ser um valor inteiro — para o símbolo s, que apresente peso p_s, será exatamente igual a $\log \frac{1}{p_s}$. Se esta condição não se verificar (o que quase sempre ocorre) alguns códigos tendem a ficar com um comprimento de $\log \frac{1}{p_s} + 1$.

Gallager em [Gal78] estimou um limite superior para a redundância do código binário de Huffman. Este valor é de:

$$R = p + 0,086,$$

onde p é a probabilidade do símbolo mais provável. Este resultado indica que o pior caso para o código de Huffman é quando p se aproxima de um. Nesta situação, embora o símbolo não contenha quase informação, ainda precisará ser representado por um *bit*.

A necessidade da utilização de valores inteiros para representar os códigos relativos aos símbolos, embora possa parecer insignificante, é um aspecto que faz com que a codificação de Huffman seja pior em relação ao resultado apresentado por outros métodos estatísticos, tais como o método Aritmético [WNC87], que conseguem representar os símbolos com uma quantidade fracionária de *bits*. Este aspecto é agravante quando a probabilidade do símbolo mais provável se aproxima de um.

Os algoritmos de compressão envolvendo o método de Huffman freqüentemente modelam os dados assumindo uma distribuição independente para os símbolos. Se este realmente é o caso, à medida que o comprimento do texto tende a infinito, a compressão atinge a entropia da fonte [Sto88]. Entretanto, conforme já mencionado, dificilmente a fonte é independente. Para ilustrar esta idéia, considere o exemplo que segue.

Exemplo 3.8 Seja o seguinte texto fonte: *aaaaaabbbbbbccccccc*. Observa-se que todos os símbolos do alfabeto $S := \{a, b, c\}$ têm o mesmo peso de 1/3. Neste caso, tem-se uma distribuição uniforme, não sendo possível, numa modelagem independente, efetuar compressão alguma.

Percebe-se, entretanto, que a distribuição dos símbolos no texto, apesar de uniforme, não é independente, pois os símbolos tendem a ocorrer em carreira.

Para o método trabalhar com fontes de mais alta ordem, tal qual a que gerou o texto do exemplo 3.8, algumas alternativas se colocam. Uma delas é a consideração de alfabetos que são n-uplas do alfabeto original. Assim, no exemplo, para uma modelagem de ordem 1, ao invés do alfabeto S, deveria se utilizar o alfabeto $S' := \{aa, ab, ac, bb, ba, bc, cc, ca, cb\}$, formado por duplas de símbolos. Uma desvantagem desta abordagem é que a modelagem considera as n-uplas como ocorrendo independentemente, e não explora a interdependência entre elas. Este aspecto só será minimizado com a consideração de n-uplas cada vez maiores. Uma variação desta alternativa é a codificação de cadeias de símbolos, ao invés dos símbolos individualmente. O seu uso entretanto, exige um qualificação das palavras no texto, o que para muitas fontes, pode ser inexequível.

Outra alternativa é a construção de uma árvore de Huffman para cada estado de um modelo markoviano. Se a fonte é de ordem zero, possui apenas um estado, sendo necessária apenas uma árvore de Huffman para representar o código. Uma fonte de ordem 1, como a determinada por S', deve gerar $|S|$ árvores de Huffman; assim, a codificação do próximo símbolo será feita com base na árvore que está representando o símbolo que o precedeu. Genericamente, $|S|^n$ árvores de Huffman devem ser mantidas e a codificação do próximo símbolo deve utilizar a árvore que representa os n símbolos que o precederam.

A grande desvantagem dessas abordagens é a enorme capacidade de armazenamento exigida, que cresce em proporção exponencial. Isto porque, uma vez que a quantidade de memória requerida normalmente é de $O(|S|)$, se considerarmos alfabetos formados de n-uplas, a memória necessária será de $O(|S|^n)$. Se um conjunto de árvores é mantido para expressar os estados de um modelo markoviano de ordem n, a memória necessária será de $O(|S|^{n+1})$. Muitos autores, a exemplo de Cormack e Horspool [CH87], Llewellyn [Lle87], Storer [Sto88], e Bell *et alii* [BWC89], destacam esta desvantagem como um forte empecilho para a utilização do código de Huffman conjuntamente com modeladores de mais alta ordem.

Do que foi exposto, podemos inferir que, embora a codificação de Huffman seja ótima do ponto de vista teórico, em termos práticos, a inadequação da construção do seu código para representar modelos de mais alta ordem dificulta o seu uso na obtenção de melhores taxas de compressão.

Implementação

A árvore de Huffman pode ser construída com o auxílio de uma estrutura do tipo *heap*, onde será representada a lista enumerada. Esta estrutura atenderá aos requisitos de ordenação, descrito no passo 1 do algoritmo 3.6, remoção (passo 2 (a)) e inserção (passo 2 (b)) de símbolos na lista. Cada operação de inserção e remoção por símbolo é efetuada em tempo $O(\log|S|)$, enquanto a ordenação em tempo $O(|S|\log|S|)$, onde $|S|$ será considerado o tamanho do alfabeto de entrada. Portanto, a construção da árvore será feita em tempo $O(|S|\log|S|)$.

Uma interessante solução para a construção da árvore é apresentada por Even em [Eve79] e por Zipstein em [Zip90], que sugerem a utilização de duas listas. A primeira lista conterá os símbolos fonte na ordem não-decrescente dos pesos. A segunda lista, que no início estará vazia, servirá para armazenar os símbolos intermediários. À medida que um símbolo intermediário é gerado, será armazenado no final da segunda lista. Como o peso de um símbolo intermediário é um resultado da soma dos menores pesos, a segunda lista manterá sempre uma ordem não-descrescente. A construção da árvore será feita pela retirada dos dois menores símbolos dentre os dois primeiros de cada uma das duas listas. A ordenação será feita em $O(|S|\log|S|)$. A remoção e inserção de cada símbolo é feita em $O(1)$ e portanto a escolha completa dos símbolos é feita em $O(|S|)$.

Por apresentar um código de prefixo, tanto a codificação quanto a decodificação de cada símbolo no texto serão realizadas em tempo real. O processo utilizado é o mesmo descrito na seção 2.1.1, p. 14, que apresenta as árvores de prefixo.

A codificação da árvore, no início do texto compresso, baseou-se no algoritmo para a representação da árvore de prefixo, fornecido na seção 2.1.1, p. 15.

Uma das desvantagens do método é gerar códigos variáveis, o que exige uma manipulação a nível de *bit*. Siemiński em [Sie88] propõe uma "decodificação" por blocos fixos, através de autômatos finitos, que evita a manipulação de *bits*. Assim, adquire um ganho no tempo de processamento, às custas de aumentar a quantidade de memória. Com base nesta idéia, Zipstein em [Zip90] propõe a "codificação" por blocos fixos. Hirschberg e Lelewer em [HL90] apresentam uma "decodificação" que, inversamente, sacrifica o tempo para utilizar a menor quantidade de memória possível.

Considerando-se N como o comprimento do texto fonte, o processo completo de compressão e descompressão são realizados em tempo $O(N)$. A quantidade de memória requerida pelas estruturas é de $O(|S|)$.

3.1.3 Método de Huffman Dinâmico (1973)

A utilização do código de Huffman, que tivesse por base as freqüências dos símbolos ocorrendo no texto fonte, necessitava de duas leituras sobre o texto, tornando a construção do código *off-line*. Motivados por esta deficiência, que de certa forma impedia o uso do código em aplicações que exigiam um algoritmo *on-line*, a exemplo da transmissão de dados, alguns autores procuraram construir códigos de Huffman dinamicamente. Numa construção dinâmica, a partir de uma única leitura sobre o texto, a árvore de Huffman é reconstruída dinamicamente de maneira a acompanhar as mudanças de pesos apresentadas pelos símbolos fonte.

Newton Faller em [Fal73] e Gallager em [Gal78], independentemente, formalizaram as principais idéias para uma construção eficiente do código dinâmico. Esta construção foi possível a partir da descoberta de uma propriedade, denominada *propriedade fraterna*, que caracteriza as árvores de Huffman. Alguns anos mais tarde, Knuth em [Knu85] apresenta em detalhes uma implementação dinâmica e fornece sugestões essenciais para o algoritmo. Este algoritmo dinâmico ficou então conhecido como FGK, em homenagem a Faller, Gallager e Knuth, que juntos o conceberam.

Recentemente Vitter em [Vit87], em uma análise sobre o algoritmo FGK, apresenta uma melhor forma de implementá-lo, de tal maneira a minimizar não somente a altura ponderada, mas também o comprimento do caminho externo e a altura da árvore. Esta característica garante a obtenção de códigos dinâmicos de altura ponderada mais próxima do código estático do que o algoritmo FGK.

Textos que apresentam genericamente os algoritmos dinâmicos de Huffman são encontrados em [LH87], [Vit87] e [Sto88].

Descrição

O método dinâmico funciona da seguinte maneira: o compressor deve manter uma árvore de Huffman correspondendo ao trecho de arquivo já processado e que expressa o conjunto de freqüências até então calculadas. Quando um novo símbolo é lido, sua freqüência aumenta de um e a árvore é atualizada de maneira a representar o novo conjunto de freqüências. Para ser possível a decodificação, o código gerado para o símbolo é o determinado pela árvore anterior à atualização. O descompressor deve manter uma árvore semelhante a do compressor e que deve ser atualizada com base no mesmo algoritmo. Assim, ao receber o código relativo ao símbolo, este é reconhecido e a árvore é reconstruída devido ao acréscimo da sua freqüência.

O processo descrito acima estabelece que ao codificar/decodificar o k-ésimo

símbolo da entrada, a árvore utilizada pelo compressor/descompressor é uma árvore de Huffman válida para a cadeia de caracteres formada pelos primeiros $k-1$ símbolos lidos. Após o reconhecimento do símbolo, a árvore é atualizada simultaneamente pelos dois processos, mantendo-se uma sincronia.

A grande dificuldade do método é a atualização da árvore. Uma abordagem imediata exigiria uma nova construção sempre que um novo símbolo fosse lido; o tempo gasto não compensaria a sua utilização. Faller em [Fal73] e Gallager em [Gal78] identificaram uma propriedade para as árvores de Huffman que permite uma atualização eficiente das mesmas. Esta propriedade foi chamada de *propriedade fraterna*[2] e passaremos a expô-la.

Definição 3.9 [Propriedade Fraterna] Seja uma árvore binária cheia e ponderada, com pesos não-negativos e no máximo um peso de valor zero. Uma *enumeração fraterna* dos seus nodos deve satisfazer às seguintes propriedades:

- Os pesos ocorrem em ordem não-decrescente;
- Cada nodo é adjacente ao seu irmão.

Diz-se que a árvore tem a *propriedade fraterna* se admite uma enumeração fraterna.

A figura 3.12 apresenta uma árvore de Huffman cujos nodos estão enumerados de maneira fraterna. O número ao lado dos nodos indica a ordem da enumeração. É fácil verificar que para $1 \leq i < |S|$, onde $|S|$ é o tamanho do alfabeto, os nodos de ordem $2i-1$ e $2i$ devem ser irmãos. A enumeração dos nodos corresponde naturalmente à ordem em que os símbolos foram sendo escolhidos na construção do código de Huffman. Este fato será demonstrado a seguir.

Teorema 3.10 *Uma árvore binária ponderada é de Huffman se e somente se apresenta a propriedade fraterna.*

Demonstração. Indução na quantidade de nodos da árvore.

Inicialmente vamos demonstrar que uma árvore binária que apresenta a propriedade fraterna é uma árvore de Huffman. Considere uma árvore binária ponderada que apresenta a propriedade fraterna e contém n nodos. Os dois primeiros nodos, p_1 e p_2, têm peso mínimo e necessariamente são nodos folhas, pois se não o fossem, seus nodos filhos teriam pesos menores que os seus e não valeria a propriedade fraterna (observe que estamos usando fortemente o fato de que apenas

[2]Do inglês, *sibling property*.

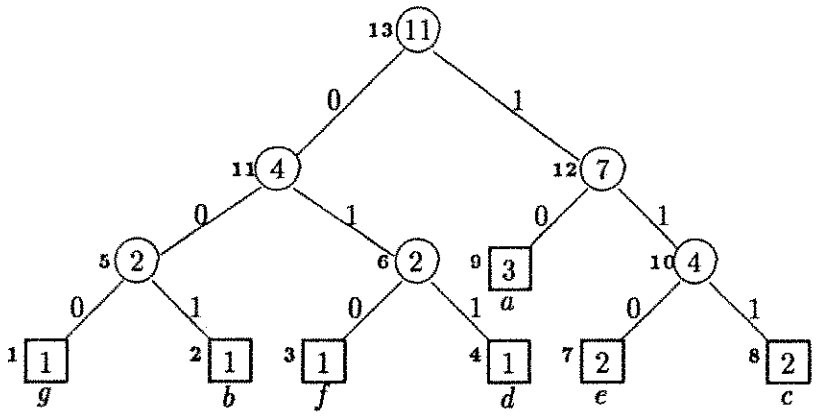


Figura 3.12: Árvore inicial de Huffman para o exemplo 3.11.

um peso ter valor zero). Remova da árvore estas duas folhas. A árvore resultante de $n-2$ nodos, juntamente com a nova enumeração, continua mantendo a propriedade fraterna. Por hipótese de indução, a nova árvore é de Huffman. Evidentemente, a árvore original também é de Huffman.

Iremos agora demonstrar que toda árvore de Huffman apresenta a propriedade fraterna. Sabe-se inicialmente, pelo teorema 3.6, que toda árvore de Huffman é cheia. Remova da árvore as duas folhas de menores pesos, p_1 e p_2. Por hipótese de indução, esta árvore de $n-2$ nodos apresenta a propriedade fraterna. Considere um enumeração para esta árvore como a indicada pela definição 3.9. Acrescente no início desta enumeração p_1 e p_2. Esta nova enumeração mostra que a árvore original tem a propriedade fraterna. □

Observe que foi utilizado na primeira parte da demonstração do teorema 3.10 o fato de que apenas um peso tem valor zero. O teorema é verdadeiro sem esta restrição, entretanto a sua prova é trabalhosa. Como veremos adiante, a adoção desta restrição não terá importância, pois estaremos utilizando sempre o conjunto de pesos adotado na definição 3.9.

A propriedade fraterna possibilita uma atualização eficiente da árvore porque estabelece um local adequado para que o símbolo com a nova freqüência possa ficar. Vamos ilustrar esta atualização com base no seguinte exemplo:

Exemplo 3.11 Considere que o texto: $aebfcdeagac\ldots$ esteja sendo compresso.

A árvore de Huffman construída até o momento está expressa na figura 3.12; os valores internos às folhas identificam a sua freqüência no texto sendo compresso. Se o próximo símbolo a ser lido for a letra b, a árvore deve ser atualizada.

Inicialmente, observe que efetuando a troca das subárvores cujos nodos raízes possuem mesmo peso e trocando-se, da mesma forma, a ordem dos nodos raízes na enumeração fraterna, a nova árvore, com a nova enumeração, continua mantendo a propriedade fraterna. No exemplo, as subárvores 1 e 3 podem ser trocadas, e os nodos que estavam enumerados na ordem: $gbfd\ldots$, passam a ser enumerados como: $fbgd\ldots$, pela troca da ordem dos nodos g e f. Verifica-se que esta nova enumeração também é fraterna. Da mesma forma, podemos trocar as subárvores 5 e 6, ou 7 e 6 e várias outras de mesmo peso. Assim, é possível realizar a troca de uma subárvore com a última subárvore na enumeração que apresente um peso equivalente ao seu. As subárvores $1, 2, 3$ e 4 do exemplo possuem o mesmo peso de 1, sendo que a de número 4 é a maior de mesmo peso na enumeração, pois logo depois vem a subárvore 5, que tem peso 2. Uma observação importante é que as subárvores a serem trocadas devem ser disjuntas, naturalmente. Para tanto, suas raízes não podem ser parentes diretas, ou seja, uma não deve estar no caminho da outra até a raiz da árvore. De fato, como no máximo um nodo tem peso zero e os demais têm peso positivo, dois nós de mesmo peso têm sempre subárvores disjuntas, com uma única exceção: o irmão do nodo de peso zero terá peso igual ao de seu pai.

Para o exemplo, ao ser lido o novo símbolo, que equivale à letra b, o peso correspondendo a sua folha deve ser acrescido de um. Para manter a árvore ponderada, os pesos dos nodos pais, no caminho que vai da folha até a raiz, devem ser acrescidos de uma unidade. Se este procedimento for aplicado diretamente na árvore, ela perderá a propriedade fraterna e portanto não será uma árvore de Huffman válida. Uma forma eficiente de se efetuar a atualização da árvore é a seguinte: se antes de acrescentarmos o peso do nodo, trocarmos a subárvore da qual ele é raiz pela subárvore cujo nodo raiz tem o mesmo peso, mas é maior na enumeração, continuamos mantendo a propriedade fraterna. Podemos agora aumentar o peso do nodo de um, pois ele estará adjacente a uma subárvore cujo nodo raiz tem peso maior que o seu, e neste caso a ordem ficará mantida. Este procedimento de troca e acréscimo de peso deverá ser realizado para o novo pai do símbolo e para todos os demais nodos no novo caminho da folha, onde foi lido o símbolo, até a raiz.

No exemplo, inicialmente o nodo 2, relativo ao símbolo b, deve ser trocado com o nodo 4, relativo ao símbolo d, e então seu peso é acrescido de um. Posteriormente, a subárvore de seu novo pai que está numerado como 6 deve ser trocada com a do nodo 8, sendo seu peso acrescido de um. O pai do nodo 8, que é o 10, deve ter sua subárvore trocada com a subárvore cuja raiz é o nodo 11, sendo seu peso acrescido de um. Finalmente, a raiz também tem seu peso acrescido de um. A árvore atualizada para o exemplo está expressa na figura 3.13.

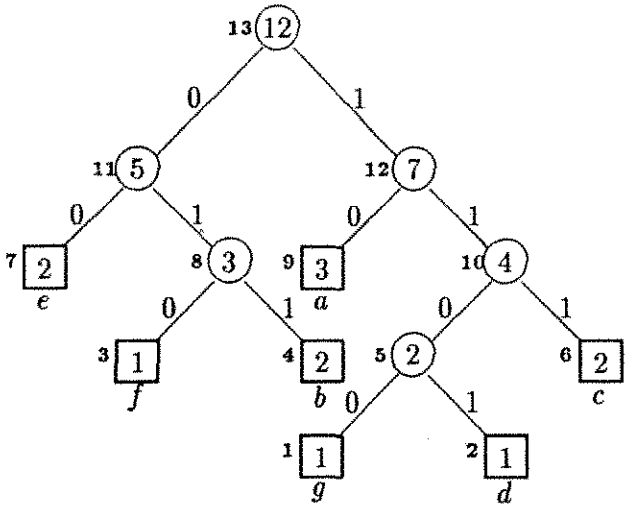


Figura 3.13: Árvore de Huffman para o exemplo 3.11 após a leitura da letra *b*.

Este procedimento de atualização dinâmica da árvore de Huffman pode ser expresso através do algoritmo da figura 3.14.

O algoritmo apresentado é eficiente porque garante a atualização da árvore de Huffman em tempo real. Ou seja, para cada símbolo lido, o número de atualizações é limitado pelo tamanho da sua altura na árvore antes do início do processo de reconstrução. Knuth em [Knu85] fornece elementos que demonstram este fato. Intuitivamente, percebe-se que para a atualização, os pesos de alguns nodos devem ser acrescidos de um. No algoritmo, antes de se efetuar este acréscimo de peso, os nodos são trocados. O nodo é trocado com um nodo maior em ordem. Pela propriedade fraterna, nodos de maior ordem "tendem" a ficar num nível da árvore mais próximo da raiz. Assim, a quantidade de nodos a percorrer fica proporcional à quantidade de nodos do caminho original. A implementação fornecida por Knuth, a ser abordada em detalhes, permite que com o auxílio de apontadores a troca entre as subárvores possa ser feita em tempo constante e portanto a atualização continua sendo em tempo real.

No exemplo 3.11, qual seria o procedimento se fosse lido um novo símbolo, por exemplo *h*, que ainda não está representado na árvore? Inicialmente, é preciso ressaltar que, independente da estrutura da árvore inicial, a partir das atualizações, a árvore dinâmica irá se modificar de maneira a expressar adequadamente a estrutura do texto. Uma alternativa seria considerar uma árvore que já contenha todos os símbolos do alfabeto, com pesos todos iguais, no início do processo. À

1. Considere que a raiz da árvore tem pai nulo (*nil*).
2. Seja n o nodo folha correspondente ao símbolo lido.
3. Enquanto $n \neq nil$ faça
   (a) Troque a subárvore em que n é a raiz pela subárvore cuja raiz apresenta o mesmo peso mas é a maior na enumeração fraterna.
   (b) Acrescente 1 ao peso de n.
   (c) Substitua n pelo seu pai.

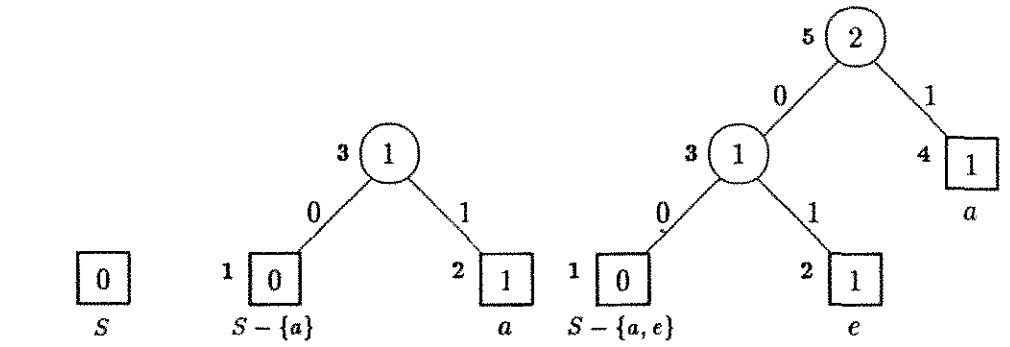
Figura 3.14: **Algoritmo para a atualização dinâmica da árvore de Huffman.**

medida que fosse sendo lido um símbolo, este iria sendo promovido e seu código iria diminuindo. Knuth em [Knu85] propôs uma implementação mais adequada que considera uma folha, chamada de *0-nodo*, porque seu peso é de 0, para representar os símbolos do alfabeto que ainda não tenham sido lidos. Quando um novo símbolo é lido, é preciso fazer com que o descompressor tenha conhecimento de qual é o símbolo. Assim, juntamente com o código relativo ao *0-nodo*, é enviado um outro código informando qual é o símbolo. Na atualização da árvore, é preciso fazer com que ela represente o novo símbolo. Assim, o *0-nodo* origina uma subárvore em que um dos filhos é um outro *0-nodo* e seu nodo irmão relaciona-se ao novo símbolo.

A figura 3.15 ilustra as árvores iniciais para os primeiros símbolos lidos do exemplo 3.11. Denotamos S como o alfabeto fonte sendo considerado.

Com a inclusão do *0-nodo*, a atualização da árvore ficará mais delicada, pois um pai terá mesmo peso que o filho. Neste caso, pelo que foi exposto no início desta seção, a troca entre pai e filho deve ser evitada. Apresentamos na figura 3.16 o algoritmo completo de atualização dinâmica, que é conhecido como algoritmo FGK.

Vitter em [Vit87] apresenta uma algoritmo dinâmico que não somente minimiza a altura ponderada da árvore, mas também minimiza a altura e o caminho externo da árvore. Como a altura da árvore é sempre a menor possível, isto significa que o código gerado para o próximo símbolo do texto fonte não será desnecessariamente grande. A base do algoritmo proposto está na manutenção



(a) Árvore inicial. (b) Árvore após a leitura do primeiro símbolo, *a*. (c) Árvore após a leitura do segundo símbolo, *e*.

Figura 3.15: Árvores iniciais para os primeiros símbolos lidos do exemplo 3.11.

de uma enumeração fraterna para a árvore que deve obedecer a estrutura visual da mesma. Ou seja, os nodos devem ser enumerados da esquerda para a direita, e as folhas devem vir antes dos nodos internos. Esta enumeração especial não é necessariamente a obtida para o algoritmo FGK. Esta modificação proposta por Vitter ao algoritmo dinâmico relaciona-se com as observações feitas por Schwartz [Sch64] para o algoritmo estático, e que foram expostas na seção 3.1.2, p. 40.

Avaliação

Podemos destacar as seguintes vantagens da modalidade dinâmica do método de Huffman:

- A não necessidade de se efetuar duas leituras sobre o texto fonte, possibilitando um algoritmo *on-line*.
- A não necessidade de se enviar a árvore de Huffman no arquivo comprimido, bastando que o compressor/descompressor sigam o mesmo algoritmo e possam portanto trabalhar independentemente.
- Possibilidade de uma maior adaptação em relação a mudanças nas características do texto fonte.

As duas primeiras vantagens são peculiares a qualquer método cujo funcionamento está baseado numa modelagem dinâmica, a terceira vantagem merece explicações. Acredita-se que o algoritmo dinâmico apresenta a característica de se adaptar a mudanças na estrutura do texto fonte, isto porque à medida que

1. Considere que a raiz da árvore tem pai nulo (*nil*).
2. Seja n o nodo folha correspondente ao símbolo lido.
3. Se n é o *0-nodo* então
   (a) Substitua o *0-nodo* por uma subárvore com dois filhos.
   (b) Enumere os nodos na ordem não-decrescente: filho *0-nodo*, filho relativo ao novo símbolo, nodo raiz.
   (c) Considere n como sendo o irmão do *0-nodo*.
4. Se n é o irmão do *0-nodo* então
   (a) Troque a folha n com a "folha" de mesmo peso que é a maior na enumeração.
   (b) Acrescente 1 ao peso de n.
   (c) Substitua n pelo seu pai.
5. Enquanto $n \neq nil$ faça
   (a) Troque a subárvore em que n é a raiz pela subárvore cuja raiz apresenta o mesmo peso, mas é a maior na enumeração.
   (b) Acrescente 1 ao peso de n.
   (c) Substitua n pelo seu pai.

Figura 3.16: **Algoritmo FGK, para a atualização dinâmica da árvore de Huffman.**

um símbolo fonte ocorre, ele irá sendo promovido na árvore. Entretanto, à proporção que o comprimento do texto tende à infinito, pode-se dizer que a estrutura apresentada pela árvore de Huffman tende a se estabilizar.

Alguns autores, explorando este potencial do método, apresentaram estratégias de melhor adaptação em relação às mudanças no texto. Gallager em [Gal78] introduz um parâmetro α responsável pelo decremento dos pesos dos nodos, em intervalos de tempo N. A depender dos valores de α e N, o método se adapta mais ou menos rapidamente a mudanças no texto. Knuth em [Knu85] introduz o conceito de uma *janela* deslizante sobre o texto fonte. Desta forma, mantém uma árvore de Huffman que expressa os pesos calculados dentro da janela. Quando o próximo símbolo é lido ele é inserido no final da janela e seu peso é acrescido de um; ao mesmo tempo, o primeiro símbolo da janela deve abandoná-la e seu peso deve ser decrescido de um. Cormack e Horspool em [CH84] propõem a atualização livre da árvore a partir do incremento ou decremento de "qualquer" valor no peso dos nodos, e não somente de um. Para que isto seja possível, apresentam um algoritmo que é semelhante ao FGK, somente que, ao invés de efetuarem a troca do nodo apenas uma vez, conforme descrito no passo 3 (a) do algoritmo 3.14, permitem que o nodo seja trocado várias vezes até que venha a ocupar a posição correta, estabelecida pelo novo peso e de forma a obedecer a propriedade fraterna. Embora estas diversas formas de adaptação sejam interessantes, a dificuldade está na escolha dos valores dos parâmetros introduzidos, que devem variar, a depender do tipo dos dados.

Apesar do algoritmo dinâmico apresentar significativas vantagens em relação ao algoritmo estático, seria necessário determinar quanto perde ou ganha o resultado alcançado pelo algoritmo FGK em relação ao resultado exibido pelo algoritmo estático, se ele fosse aplicado ao mesmo texto fonte.

Considerando-se $|S|$ como o tamanho do alfabeto fonte e tomando-se E como o resultado, em *bits*, obtido pelo algoritmo estático na compressão de um texto fonte que apresente comprimento de N símbolos, Vitter em [Vit87] demonstrou que o algoritmo FGK, no máximo, obtém quase duas vezes mais o resultado do algoritmo estático, ou seja, FGK é $\approx 2 * E + N$ *bits* no pior caso. O algoritmo proposto por Vitter, por sua vez, tem um melhor desempenho, pois, no máximo, ele excede de um *bit* o comprimento para o código de cada símbolo. Ou seja, no pior caso ele é $< E + N$ *bits*. Com a obtenção deste resultado, Vitter demonstra que o seu algoritmo obtém o melhor resultado possível dentre os demais métodos dinâmicos de Huffman. Intuitivamente, este fato é explicado, pois, uma vez que o algoritmo de Vitter também minimiza a altura e o caminho externo da árvore, a codificação de um novo símbolo será sempre realizada com o menor código possível.

Esta diferença de resultados, embora significativa, na prática, só é sentida quando a compressão é efetuada sobre textos de comprimento pequeno, pois à medida que o comprimento do texto tende a infinito os desempenhos dos algoritmos mencionados se equiparam.

Implementação

Vitter em [Vit89] apresenta uma implementação em PASCAL para o algoritmo proposto em [Vit87]. Esta implementação executa em tempo real. Isto é, a atualização de um símbolo que se encontra num nível l da árvore é feita em tempo $O(l)$.

Uma implementação detalhada, em pseudocódigo, do algoritmo FGK é fornecida por Knuth em [Knu85]. Foi esta implementação que adotamos para a realização dos experimentos e passamos a expô-la.

Knuth propõe uma representação para a árvore de Huffman que permite a sua atualização em tempo real. Esta representação se utiliza da ordem para os nodos estabelecida pela propriedade fraterna. A enumeração fraterna dos nodos da árvore corresponde às próprias posições físicas das estruturas de dados utilizadas para armazenar as informações relativas aos nodos. Assim, por ex., sendo $|S|$ o tamanho do alfabeto, a quantidade total de nodos da árvore será $2|S|-1$; o nodo raiz da árvore ocupará sempre esta posição.

Cada nodo da árvore possui três apontadores. Um apontador para o nodo pai; um apontador para o filho esquerdo — pela propriedade fraterna, sendo o filho esquerdo $2j-1$, o direito será $2j$; um apontador para um bloco a que pertence o nodo. Nodos de mesmo peso pertencem ao mesmo bloco.

Os blocos estarão ordenados pelo peso em ordem não-decrescente e formarão uma lista circular duplamente encadeada. Cada bloco conterá três apontadores: o apontador esquerdo identifica o bloco imediatamente anterior; o apontador direito identifica o bloco imediatamente posterior; um último apontador servirá para indicar o nodo da árvore que contenha o peso do bloco e que se encontre em maior ordem na enumeração.

A árvore está ligada à lista de blocos pelos pesos dos nodos e esta por sua vez está ligada a árvore apontando para o último nodo que contenha o peso que o bloco representa. Estas estruturas permitem a atualização da árvore em tempo real, e ocupam espaço $O(|S|)$.

Para manter a enumeração fraterna, sempre que o *0-nodo* origina uma nova subárvore (passo 3 do algoritmo 3.16), os novos nodos criados ocuparão as próximas posições disponíveis das estruturas na ordem decrescente. Assim, se o *0-nodo* anteriormente ocupava a posição n da árvore, a subdivisão fará com que o novo

0-nodo ocupe a posição $n-2$, seu irmão a posição $n-1$ e seu pai a posição n. A troca das subárvores (passo 4 (a) e 5 (a)) consiste em: trocar o conteúdo dos nodos; atualizar o apontador para os filhos destes nodos e o apontador dos pais dos mesmos filhos.

Além da compressão de caracteres, implementamos o método de Huffman dinâmico efetuando compressão de palavras, i.é, cadeias de caracteres. Uma vez que o conjunto de palavras no texto é sempre bem maior que o alfabeto de caracteres, precisou-se manter um dicionário de palavras de tamanho limitado. O índice da palavra no dicionário foi o utilizado para a sua codificação. Em verdade, foram mantidas três árvores de Huffman; uma que codifica o índice da palavra no dicionário, outra que codifica os caracteres da nova palavra, e outra que codifica o seu comprimento. Devido ao grande número de palavras e para uma melhor adaptação às mudanças de características no texto, utilizou-se o conceito de *janela* sobre o texto. A manutenção do dicionário foi feita a partir de uma *função de espalhamento*, e portanto a determinação das palavras pôde ser feito em tempo constante.

3.1.4 Método Aritmético (1963)

O princípio do método aritmético encontra-se no código proposto por Shannon em [SW49]. Este código, descrito na seção 3.1.1, p. 35, tem por base os pesos cumulativos dos símbolos, estabelecido pela fórmula (3.2). Aperfeiçoando esta idéia, Elias propôs um método que foi apresentado por Abramson em [Abr63, pp. 61–62]. O princípio seria aplicar um processo recursivo ao código de Shannon, objetivando encontrar o peso cumulativo para o próprio texto fonte de N símbolos. Isto seria feito a partir dos pesos cumulativos dos símbolos individualmente e do peso cumulativo do texto formado por $N-1$ símbolos. Este processo daria origem a uma representação do texto fonte por um valor numérico no intervalo real de $[0,1)$. Quanto menos dígitos este valor possuísse, maior seria a compressão alcançada.

O grande problema desta abordagem estava na precisão requerida para a representação do valor numérico que aumentava com o comprimento do texto. Alguns anos mais tarde, alguns autores, a exemplo de Pasco [Pas76], Rissanen [Ris76], Rubin [Rub79], e Rissanen e Langdon [RL79], objetivando vencer esta e outras dificuldades de ordem prática, forneceram melhorias para o método. Estes dois últimos autores ainda generalizaram e caracterizaram a família dos códigos aritméticos. Recentemente, Witten *et alii* em [WNC87] apresentaram uma implementação em linguagem C que não só responde às dificuldades práticas, mas acrescenta algumas características ao método que permitem, por exemplo,

sua utilização *on-line*. Além disso, estabelecem claramente a separação entre a codificação aritmética e o modelo de fonte de dados em que ela se baseará. Torna-se então possível a existência de um método em que é indiferente a utilização de uma modelagem estática ou de um modelagem dinâmica dos dados, de uma modelagem de ordem 0 ou de alta ordem.

Devido a esta característica de permitir uma distinção entre o modelo de fonte e a codificação, o código aritmético pode ser utilizado conjuntamente com processos adaptativos de modelagem markoviana na composição de poderosos compressores. Os métodos apresentados por Cleary e Witten [CW84], Cormack e Horspool [CH87], Llewellyn [Lle87] e vários outros métodos mencionados por Bell *et alii* [BWC89] utilizam esta abordagem.

Embora o método aritmético apresente claras vantagens que o colocam como um sério competidor para os demais métodos estatísticos e conseqüentemente para o método de Huffman [Huf52], há um reconhecimento de que este método e suas potencialidades ainda não foram largamente difundidos [WNC87].

Textos introdutórios sobre esta forma de codificação são encontrados em [Lan84], [LH87] e [WNC87].

Descrição

O método aritmético irá representar o texto fonte por um valor numérico no intervalo real de $[0, 1)$. O objetivo será encontrar um número que apresente uma pequena quantidade de dígitos. Para tanto, supõe uma fonte de dados bem estabelecida, em que os símbolos do alfabeto ocorrem no texto com um peso determinado.

Inicialmente, considera-se o intervalo de $[0, 1)$ para a representação do texto. À medida que o texto vai sendo codificado, este intervalo vai diminuindo e a quantidade de dígitos para representá-lo vai aumentando. A leitura de um novo símbolo reduz o intervalo a um valor que depende de seu peso. Quanto maior o peso, maior será o intervalo obtido, o que não aumenta muito a quantidade de dígitos para representar o texto. Se o símbolo tem peso pequeno, o intervalo obtido será pequeno e a quantidade de dígitos para representar o texto aumenta.

Vamos ilustrar este método através do exemplo seguinte. Para facilidade de apresentação, estaremos utilizando a aritmética decimal.

Exemplo 3.12 Considere o seguinte exemplo de texto fonte: *bacb#*.

A cada símbolo do alfabeto fonte deve ser atribuída uma faixa do intervalo inicial $[0, 1)$; esta faixa é estabelecida com base nos pesos cumulativos dos símbolos,

símbolo (s)	peso (p)	peso cumulativo (P)	faixa $[i, f)$
a	0,2	0,0	[0,0; 0,2)
b	0,4	0,2	[0,2; 0,6)
c	0,2	0,6	[0,6; 0,8)
#	0,2	0,8	[0,8; 1,0)

Tabela 3.4: Tabela dos símbolos para o exemplo 3.12

Figura 3.17: Ilustração da subdivisão do intervalo $[0, 1)$ entre os símbolos do alfabeto para o exemplo 3.12.

que são calculados a partir da relação (3.2). A tabela 3.4 apresenta os símbolos com os respectivos pesos e faixas. Observa-se que os símbolos não necessariamente precisam estar em uma ordem. A quarta coluna da tabela, que expressa as faixas, indica que à letra a cabem os primeiros 20% do intervalo, à letra b, cabe a faixa que vai de 20% a 60%, e assim sucessivamente. A figura 3.17 ilustra esta subdivisão do intervalo real nos símbolos do alfabeto. É importante destacar que a subdivisão do intervalo entre faixas para os símbolos é necessária para ser possível a decodificação.

No início da codificação o texto fonte é representado pelo intervalo $[0, 1)$. Após a leitura do símbolo b, de acordo com o seu peso, o intervalo deve ter sua amplitude reduzida para 40%. De acordo com a faixa designada para b, os novos limites do intervalo deverão estar entre 20% e 60% do intervalo anterior. Sendo assim, o intervalo será reduzido a $[0,2;\ 0,6)$. O próximo símbolo, que equivale à letra a, irá reduzir este intervalo para os primeiros 20%, resultando o intervalo $[0,2;\ 0,28)$. A figura 3.18 ilustra este início de codificação. O símbolo seguinte, que corresponde à letra c, irá reduzir o intervalo anterior para 20%, nos limites de 60% a 80%, e portanto o intervalo ficará $[0,248;\ 0,264)$. O símbolo b, que vem depois, irá reduzir o intervalo para $[0,2512;\ 0,2576)$ e finalmente, o último símbolo # reduzirá o intervalo para os últimos 20%, resultando $[0,25632; 0,2576)$. O número para representar o texto fonte será qualquer valor neste intervalo. Escolhemos então o valor $0,257$, equivalente a três dígitos decimais. A entropia

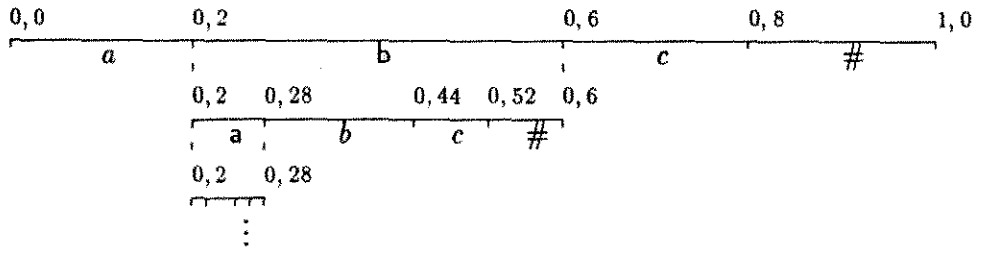


Figura 3.18: Ilustração da codificação aritmética para o texto *ba* ... do exemplo 3.12.

para esta fonte será de:

$$H(S) = 3(0,2 * \log_{10} \frac{10}{2}) + 0,4 * \log_{10} \frac{10}{4} = 0,578558,$$

e a quantidade de dígitos para representar o texto fonte será: $5 * H(S) = 2,89279$. Portanto, a redundância do código é de $0,10721$.

Para decodificar o valor numérico que representa o texto, o descompressor deverá seguir o mesmo caminho que o compressor. Assim, inicialmente considera que o texto é representado pelo intervalo real $[0,1)$. O valor numérico de $0,257$ indica que o primeiro símbolo deverá estar no intervalo de $[0,2;\ 0,6)$ e portanto corresponde à letra *b*. Seguindo os mesmos passos que o compressor, o descompressor reduz o intervalo a este novo valor e novamente, com base nos pesos dos símbolos e no valor numérico, determina que o segundo símbolo pertence ao intervalo $[0,2;\ 0,28)$, que corresponde à letra *a*. O próximo intervalo a que o número pertence será $[0,248;\ 0,264)$, que corresponde à letra *c*. Este processo continua até que todos os símbolos tenham sido decodificados.

Após se decodificar o último símbolo, que equivale à letra #, chega-se ao intervalo de $[0,25632;\ 0,2576)$. Observa-se entretanto, que somente o valor numérico fornecido não indica ser este o fim do texto fonte, o que permite ao descompressor continuar o processo, reduzindo o intervalo anterior para o novo intervalo de $[0,256576;\ 0,257344)$ a que pertence o número e que equivale à letra *b*. Este processo pode se repetir indefinidamente, já que considera-se uma escala real. Um exemplo que elucida bem este problema é o do seguinte texto: *aaaa*..., que independente do número de $a's$, pode ser codificado com o valor $0,0$.

Para resolver o problema anterior, pode-se pensar em fornecer ao descompressor o comprimento do texto fonte. Esta solução não é muito interessante pois restringe o método a um funcionamento *off-line*. Outra alternativa, é acrescentar

ao fim do texto fonte um símbolo indicador de final de texto. No exemplo anterior, o símbolo # tem esta atribuição. Ao reconhecê-lo o descompressor finaliza o processo.

Observa-se, a partir do exemplo apresentado, que o código aritmético admite uma definição recursiva. Isto porque a determinação do intervalo que representa um texto de N símbolos, é feita com base no intervalo que representa o texto de $N-1$ símbolos, e do intervalo do n-ésimo símbolo.

Verificou-se que a obtenção de um novo intervalo é feita a partir de uma redução no intervalo atual, que depende do peso do novo símbolo lido do texto. Para esta redução são necessárias duas recorrências. A primeira recorrência, que determina o início do novo intervalo, tem por base o início e a amplitude do intervalo atual, e o peso cumulativo do novo símbolo. A segunda recorrência, que determina a amplitude do novo intervalo, baseia-se na amplitude do intervalo atual e no peso do novo símbolo. Com base neste dois valores o final do intervalo pode ser determinado. Este fato nos leva à seguinte definição dos códigos aritméticos.

Definição 3.13 [Código Aritmético] Considere um alfabeto de S símbolos, onde cada símbolo s tem peso p_s e peso cumulativo P_s. Estes dois valores estabelecem um intervalo $I(s) := [i_s, f_s)$ para s, onde $i_s := P_s$ e $f_s := i_s + p_s$.

Seja t um texto sobre S. Define-se o *código aritmético* para t como sendo um valor numérico num intervalo $I(t) = [i_t, f_t)$ que obedece à seguinte recursão:

$$I(t) := \begin{cases} [0,1), & \text{se } t = \epsilon, \\ I(t') \circ I(s), & \text{se } t = t's, \quad s \in S,\ t' \in S^*. \end{cases}$$

Considerando-se $A := f - i$ como sendo a amplitude de um intervalo $I := [i, f)$, define-se a operação $\circ$ para a determinação de $I(t)$ através das seguintes relações:

$$\begin{array}{lrcl} 1. & i_t & = & i_{t'} + A_{t'} * P_s; \\ 2. & A_t & = & A_{t'} * p_s. \end{array}$$

Com base em 1 e 2, define-se $f_t = i_t + A_t$.

Apresentamos na figura 3.19 um algoritmo para a codificação aritmética que segue a definição acima. Para esclarecer melhor este processo, apresentamos novamente na figura 3.20 a codificação do texto do exemplo 3.12.

Conforme observado na apresentação do exemplo 3.12, a decodificação deve obedecer a uma recursão, seguindo os mesmos passos realizados pela codificação. Entretanto, as operações devem ser efetuadas de maneira inversa. Em cada etapa

1. Seja S um alfabeto, onde cada símbolo s tem associado um peso p_s e um peso cumulativo P_s.
2. Seja t um texto formado por símbolos de S, $t \in S^*$.
3. Seja $I := [i, f)$ o intervalo para o texto t, e $A := f - i$ a sua amplitude.
   $I := [0, 1)$, $A := 1$.
4. Enquanto $t \neq \epsilon$ faça
   (a) Obtenha s de t.
   $t = st'$, $s \in S, t' \in S^*$.
   (b) Reduza o intervalo I:
      i. $i := i + A * P_s$.
      ii. $A := A * p_s$.
      iii. $f := i + A$.
   (c) $t := t'$.
5. Associar a t os dígitos da fração mais curta em I.

Figura 3.19: **Algoritmo para a Codificação Aritmética.**

Texto original: *bacb#*

Passo	Símbolo lido	Redução	Intervalo corrente
1	*b*	$I = [0;1) \circ [0,2;\ 0,6)$	
		$i = 0 + (1 * 0,2)$	
		$A = 1 * 0,4$	$I = [0,2;\ 0,6)$
2	*a*	$I = [0,2;\ 0,6) \circ [0,0;\ 0,2)$	
		$i = 0,2 + (0,4 * 0,0)$	
		$A = 0,4 * 0,2$	$I = [0,2;\ 0,28)$
3	*c*	$I = [0,2;\ 0,28) \circ [0,6;\ 0,8)$	
		$i = 0,2 + (0,08 * 0,6)$	
		$A = 0,08 * 0,2$	$I = [0,248;\ 0,264)$
4	*b*	$I = [0,248;\ 0,264) \circ [0,2;\ 0,6)$	
		$i = 0,248 + (0,016 * 0,2)$	
		$A = 0,016 * 0,4$	$I = [0,2512;\ 0,2576)$
5	#	$I = [0,2512;\ 0,2576) \circ [0,8;1,0)$	
		$i = 0,2512 + (0,0064 * 0,8)$	
		$A = 0,0064 * 0,2$	$I = [0,25632;\ 0,2576)$

Texto codificado: $0,257$

Figura 3.20: Codificação para o texto do exemplo 3.12.

1. Seja S um alfabeto, onde cada símbolo s tem associado um peso p_s e um peso cumulativo P_s.
2. Seja t o texto resultante da decodificação e # o indicador de final de t. $t := \epsilon$.
3. Seja v o valor numérico que representa o texto t.
4. Seja $I := [i, f)$ o intervalo para o texto t, e $A := f - i$ a sua amplitude. $I := [0, 1)$, $A := 1$.
5. Enquanto $s \neq \#$ faça
   (a) Encontre s tal que: $i_s \leq (v - i)/A < f_s$.
   (b) Se $s \neq \#$ então
      i. Reduza o intervalo I:
         - $i := i + A * P_s$.
         - $A := A * p_s$.
         - $f := i + A$.
      ii. $t := ts$.

Figura 3.21: **Algoritmo para a Decodificação Aritmética.**

da codificação, as operações que originaram o valor numérico foram as descritas no passo 4 (b) do algoritmo 3.19. Efetuando operações inversas a estas, em cada decodificação, o descompressor deverá encontrar o símbolo referente ao intervalo ao qual pertence o valor numérico. Isto é feito subtraindo-se o valor numérico do valor inicial do intervalo atual e dividindo-se este valor pela amplitude atual. Após o reconhecimento do símbolo, o intervalo deve ser reduzido, da mesma forma que no passo 4 (b). Apresentamos na figura 3.21 um algoritmo de decodificação que tem por base este processo.

Para que a descompressão seja possível, é preciso que o valor numérico resultante da compressão seja conhecido. Este fato levaria a uma decodificação do texto somente após ele ter sido completamente codificado. Pode-se entretanto, efetuar este processo de maneira *incremental*. Ou seja, sempre que possível, a codificação de um dígito relativo ao número é efetuada. Isto pode ser realizado

quando o intervalo possui limites cujos valores iniciam-se pelo mesmo dígito. Uma vez que estes dígitos não poderão ser mais modificados, o compressor poderá gerá-los. No exemplo 3.12, ao se codificar o texto *ab*, o intervalo resultará em [0,2; 0,28), que começa e termina por 2, podendo este número já ser codificado.

Demonstra-se que a decodificação de um texto pode ser feita sem ambigüidades, [RL79] e portanto, de certa forma, o código aritmético é *unicamente decodificável.* Observa-se entretanto que os conceitos apresentados no Capítulo 2 sobre as características de decodificabilidade para um código não se aplicam a este método. A maioria dos métodos atribuem códigos individualmente para os símbolos do alfabeto, e o resultado é um texto comprimido como uma *concatenação* desses códigos. Características desejáveis a esses códigos foram amplamente apresentadas no Capítulo 2. A codificação aritmética, diferentemente, a partir das pesos para os símbolos, encontra um único código, equivalente a um valor numérico, para representar todo o texto fonte. Para adequar este novo paradigma à noção de código, Rissanen e Langdon [RL79] generalizaram as características da seguinte maneira: concatenação é substituída por adição; códigos para os símbolos por números racionais com casas finitas; comprimentos inteiros por comprimentos racionais; propriedade de prefixo por ordem de magnitude.

Avaliação

O comprimento do código obtido pela codificação aritmética aproxima-se do valor da entropia para o modelo de fonte adotado.

No exemplo 3.12, a amplitude do intervalo final é de 0,00128. A quantidade de dígitos decimais para representar algum número neste intervalo é de $-\log_{10} 0,00128 = 2.89279$, que corresponde exatamente ao valor da entropia. Esta coincidência ocorre porque a amplitude do intervalo final é um valor resultante da multiplicação dos pesos dos diversos símbolos ocorrendo no texto. Isto é determinado pela relação de número 2 da definição 3.13.

Tomando-se A como sendo a amplitude do intervalo final, p_i como sendo o peso para o símbolo i, $|S|$ como sendo o tamanho do alfabeto fonte e N como sendo o comprimento total do texto, tem-se:

$$A = \Pi_{i=1}^{N} p_i.$$

Assim, a quantidade de dígitos para representar o intervalo será:

$$-\log A = -\log \Pi_{i=1}^{N} p_i = -\Sigma_{i=1}^{N} \log p_i = N(-\Sigma_{i=1}^{|S|} p_i \log p_i) = N * H(S),$$

que corresponde exatamente ao valor da entropia para o texto fonte.

Os códigos aritméticos apresentam muitas vantagens. Uma delas é a não necessidade de representar os símbolos por uma quantidade inteira de *bits*. Uma vez que o resultado da compressão não é uma concatenação de códigos individuais, isto é possível. Se um símbolo ocorre no texto com uma freqüência alta, esta característica é importante. Se por exemplo o texto: *aaa...*, *a* ocorrendo n vezes, deve ser codificado, um código estatístico, tal como o de Huffman, irá representá-lo por n *bits*; enquanto que o código aritmético irá representá-lo por uma pequena quantidade de *bits*, pois são necessárias muitas ocorrências do símbolo *a* para a redução do intervalo a um valor cujos limites apresentam mesmo dígito, e conseqüentemente para a saída de "um" *bit* no texto compresso.

A vantagem acima evidencia uma característica importante que é o fato da codificação de Huffman se constituir em um caso particular dos códigos aritméticos, em que a aritmética utilizada é de comprimento inteiro. Rissanen e Langdon em [RL79] demonstram este fato.

Outra grande vantagem deste método é a possibilidade de adequação do código a qualquer mudança no modelo de dados. Este aspecto é esclarecido pelo próprio algoritmo de codificação (3.19), que apenas necessita do conhecimento dos pesos dos símbolos para a próxima redução, independentemente destes pesos estarem mudando ou não. Rissanen e Langdon em [RL81] esclarecem este fato. Witten *et alii* em [WNC87] fornecem uma implementação em que não há diferenças entre a utilização do método com uma modelagem estática ou adaptativa.

O aspecto mencionado acima possibilita o uso do método com modeladores markovianos de mais alta ordem. Bell *et alii* em [BWC89] apresentam uma extensa lista de compressores que propõem uma modelagem adaptativa para, principalmente, uma utilização conjunta com o código aritmético, em função das vantagens apresentadas anteriormente. Dentre os principais compressores destacam-se os modeladores propostos por Cleary e Witten [CW84], Cormack e Horspool [CH87], e Llewellyn [Lle87]. A modelagem efetuada por estes métodos baseia-se no uso de autômatos probabilísticos.

O uso conjunto de modeladores de mais alta ordem com a codificação estatística vem sendo apontado como o caminho que poderá trazer resultados futuros à área de compressão. [BWC89]

Implementação

Witten *et alii* em [WNC87] propõem uma implementação detalhada que se ocupa de algumas preocupações, listadas a seguir:

- A aritmética utilizada deve ser a binária;

- O processo de codificação/decodificação deve ser incremental;
- A aritmética utilizada deve ser inteira, devendo-se evitar a ocorrência de *overflow* ou *underflow*, no cálculo dos intervalos;
- A representação do modelo de dados — símbolos com os pesos — sendo separada, deve ser eficiente para permitir um acesso rápido por parte, principalmente, do decodificador.

A seguir, iremos detalhar a implementação fornecida pelos autores. Foi esta implementação que adotamos para a realização dos experimentos do Capítulo 4.

Para a representação do modelo de dados é utilizado um vetor de freqüências cumulativas, que é atualizado com base num contador de freqüências mantido para cada símbolo. Mantém-se sempre uma diferença de 1 para as freqüências, para que não haja interpenetração dos intervalos. A freqüência total dos símbolos não pode exceder um valor máximo, e se isto ocorre as freqüências devem ser escaladas, sendo divididas por 2 para se evitar *overflow*. O vetor de freqüências cumulativas é mantido em ordem decrescente para facilitar, na decodificação, a busca dos símbolos. O contador da freqüência total é utilizado para normalizar o valor das freqüências. São apresentados dois modelos, um fixo, com pesos pré-estabelecidos e um adaptativo.

Para a representação do intervalo $[0, 1)$ existe um tipo de dado especial designado de *valor de código*. Este valor estabelece as seguintes constantes: um *valor máximo*, que identifica o maior valor de código, e a *metade* deste valor. Estas constantes representam partes do intervalo.

No início do processo, o início e o final do intervalo são representados por 0 e o *valor máximo*, respectivamente. À medida que o intervalo vai sendo reduzido, os *bits* dos dois limites podem se tornar os mesmos. Isto significa que os dois limites estão abaixo da *metade* do intervalo, neste caso é emitido o *bit* 0, ou acima, quando é emitido o *bit* 1. Desta maneira, é possível uma operação incremental. Este processo, embora esteja sempre ampliando o intervalo, não impede que ele seja reduzido o bastante para existir o *underflow*. Isto acontece quando os dois limites do intervalo se aproximam muito da metade, mas mantêm sempre *bits* distintos. A solução encontrada para este caso foi a manutenção da amplitude do intervalo em um valor tão grande quanto a *máxima freqüência*. Para que isto seja possível, o *valor de código*, que representa o intervalo, deverá ter um tamanho 4 vezes maior que a da máxima freqüência. Em outras palavras, se f é a quantidade de *bits* da freqüência e c do valor de código, vale a relação: $f \leq c - 2$.

A codificação aritmética em si, efetua para cada símbolo apenas umas poucas operações de adição e multiplicação; além disso, a quantidade de memória re-

querida é mínima. Para a atualização do modelo de dados adaptativo proposto, a manutenção da ordem no vetor de freqüências cumulativas é feita de maneira seqüencial. Considerando-se $|S|$ como o tamanho do alfabeto, esta atualização leva tempo $O(|S|)$ para cada codificação/decodificação de símbolo. O espaço ocupado pelas estruturas de dados é de ordem $O(|S|)$. É importante observar que os modelos de mais alta ordem existentes freqüentemente requerem uma grande quantidade de memória.

3.2 Métodos de Codificação por Fatores

Os métodos de codificação por fatores exibem um processo de modelagem e codificação conjunto, numa tentativa de identificar e remover variados tipos de redundância. Por isso, a afirmação de que eles "apreendem" as características das fontes de dados. A heurística que costumam utilizar ora favorece um tipo de redundância ora outros, e isto é realizado com uma boa dose de criatividade.

Basicamente, tenta-se substituir cadeias de símbolos consecutivos (fatores), que ocorrem mais de uma vez no texto fonte, por índices a uma única cópia dos fatores, que estarão presentes numa memória interna, compondo um dicionário de dados. Estes índices, por sua vez, devem ter comprimentos menores do que o dos fatores que representam. Por este motivo, estes métodos também são ditos de *substituição textual* [SS82] [FG89], ou *codificação por dicionário* [BWC89].

De acordo com a heurística de compressão apresentada acima, os métodos desta categoria se diferenciam na maneira como realizam os seguintes aspectos:

- Análise Léxica — como os fatores são reconhecidos no texto;
- Determinação do Dicionário — como os fatores são mantidos no dicionário.

Para a identificação do próximo fator no texto fonte, i.é, para a realização da *análise léxica*, os métodos, em sua totalidade, adotam a *heurística gulosa*,[3] em que são sempre reconhecidos os próximos símbolos do texto fonte que compõem o maior fator existente no dicionário. Esta estratégia não é necessariamente ótima, pois a escolha de um fator longo num momento pode excluir a possibilidade de escolha de um fator muito longo no futuro. Apesar deste fato, uma abordagem ótima iria requerer a avaliação prévia do texto fonte [BWC89]; ademais, na prática, a heurística gulosa tende a um desempenho ótimo [Sto88].

Os melhores e mais conhecidos métodos desta categoria têm um funcionamento adaptativo. Ou seja, o dicionário de dados é atualizado dinamicamente

[3] Em inglês, *greedy match heuristic*.

durante a compressão. Por sua vez, a maioria dos métodos adaptativos descende de dois métodos propostos, quase que ao mesmo tempo, por Ziv e Lempel. Um dos métodos foi apresentado em 1977 [LZ76] [ZL77] e o outro em 1978 [ZL78]. Grande parte dos métodos existentes diferencia-se destes originais apenas pela variação de alguns parâmetros, por vezes só significativa na prática. Devido às idéias pioneiras destes dois autores, esta categoria também é conhecida por *compressores Lempel-Ziv.*[4]

Estratégias para a utilização de dicionários estáticos são mencionadas por Bell *et alii* em [BWC89]. Através de um estudo teórico aprofundado sobre *métodos de substituição textual off-line*, Storer [Sto88] discute a utilização de modelos estáticos. Sobre este aspecto, um importante resultado é que a determinação de um dicionário ótimo para um dado texto vem a ser um problema NP-difícil no tamanho do texto [SS82] [FMP83].

Alguns métodos exibem uma modelagem que explora aspectos particulares de redundância. Este é o caso do método desenvolvido independentemente por Elias [Eli87] e Bentley *et alii* [BSTW86], que procura identificar adaptativamente localidades de referência no texto.

Uma importante característica para os métodos desta categoria é a exibição de um funcionamento bastante simples e rápido. Este fato deve-se tanto ao reconhecimento no texto fonte de longas cadeias, em oposição ao reconhecimento de símbolos de pequeno comprimento dos métodos estatísticos, quanto à possibilidade da geração de códigos de comprimento fixo, que acompanhem o alinhamento da palavra da máquina. Um outro aspecto motivador é que, se a heurística que esses métodos apresentam é realmente válida, eles têm a capacidade de alcançar a entropia de qualquer fonte de dados. Hansel em [Han89] demonstrou que este resultado já foi alcançado adaptativamente pelo método de Lempel-Ziv [ZL78].

Storer em [Sto88] fornece um estudo extenso sobre esta categoria de compressores, enfocando tanto o aspecto prático como o teórico. Ele propõe três métodos *on-line*, designados de *dicionário estático, deslizante e dinâmico*,[5] e apresenta implementações; realiza um estudo teórico sobre métodos *off-line*; apresenta algoritmos paralelos para uma implementação em VLSI dos três métodos propostos.

Todos os métodos a serem apresentados nesta seção exibem um funcionamento adaptativo. Estes métodos são o método de Lempel e Ziv (1977) [ZL77] e alguns dos seus principais descendentes [SS82], [FG89]; o método de Lempel e Ziv (1978) [ZL78] e seus principais descendentes [Wel84], [MW85], [Sto88]; o método de

[4] Não sabemos o porquê, mas existe um consenso em se designar esta categoria por compressores Lempel-Ziv, em oposição a Ziv-Lempel, como seria o adequado, em concordância com a ordem dos autores nos artigos.

[5] Em inglês, *static, sliding* e *dynamic dictionary*.

Elias-Bentley [Eli87] [BSTW86].

3.2.1 Método de Elias-Bentley (1986)

Elias [Eli87] e Bentley *et alii* [BSTW86], independentemente, idealizaram um método de compressão objetivando remover a redundância devido à *localidade de referência* que muitos textos apresentam.

O método tem um funcionamento adaptativo e se baseia no reconhecimento de palavras bem-definidas no texto. As palavras são mantidas numa lista, cuja atualização segue uma heurística de *busca seqüencial auto-organizável*. Nesta técnica, palavras freqüentes no texto são promovidas para o início da lista. O código resultante deste método é o índice da palavra na lista. Índices menores devem ocupar menos espaço.

O simples processo descrito acima, juntamente com uma codificação de prefixo apropriada, obtém taxas de compressão comparáveis ao método de Huffman. Nos casos em que a redundância local está presente, o desempenho é superior.

A idéia fornecida pelo método permite muitas variações. Estas variações dependem da escolha da heurística para manter as palavras freqüentes no início da lista. Com base nesta escolha muitos métodos de compressão podem ser extraídos. Bentley *et alii* [BSTW86] apresentam o método *movimento-para-frente*;[6] Elias [Eli87] apresenta dois métodos, *codificação por intervalo*[7] e *codificação por ocorrência recente*;[8] Mäkinen [Mak89] apresenta duas implementações, uma para a codificação por intervalo e outra para a *codificação por transposição*.

Descrição

Bentley *et alii* [BSTW86] e Elias [Eli87], independentemente, apresentaram dois métodos que se baseavam numa mesma heurística para a manutenção de palavras freqüentes no início de uma lista de palavras. Estes dois métodos foram chamados respectivamente de *movimento-para-frente*, do inglês, *move-to-front* (MTF), e *codificação por ocorrência recente*. Iremos no restante deste trabalho designar este método por Elias-Bentley.

O funcionamento do método Elias-Bentley é adaptativo e tem por princípio a seguinte heurística: sempre que uma palavra é reconhecida no texto ela é movida para o início da lista de palavras. O código gerado será a posição da palavra na lista antes do movimento. Quanto mais perto do início, menos dígitos deverão

[6] Em inglês, *move-to-front* (MTF).
[7] Em inglês, *interval coding*.
[8] Em inglês, *recency rank coding*.

1. Seja t um texto formado de palavras sobre um alfabeto S de símbolos, $t \in S^*$.
2. Seja L uma lista de palavras e $|L|$ o seu tamanho, $L := \epsilon$, $|L| := 0$.
3. Enquanto $t \neq \epsilon$ faça
   (a) Obtenha w de t. $t = wt'$, $w, t' \in S^*$.
   (b) Se $w \in L$ então
      i. Codifique a posição de w em L.
   (c) Senão
      i. Codifique a posição $|L| + 1$.
      ii. Codifique w.
      iii. Acrescente w a L.
      iv. $|L| := |L| + 1$.
   (d) Mova w para o início de L.
   (e) $t := t'$.

Figura 3.22: **Algoritmo para a Compressão de Elias-Bentley.**

ser necessários para a representação desta posição. Este processo mantém as palavras que estão ocorrendo no momento mais próximas do início da lista, e desta forma consegue eliminar alguma redundância local que possa existir.

Sendo um método adaptativo, o compressor deve apresentar um algoritmo que o descompressor possa seguir. Na codificação, se a próxima palavra a ser reconhecida já estiver presente na lista, o código gerado será a sua posição na lista. Caso contrário, ela deverá ser acrescentada ao final da lista e o código deverá ser esta posição, juntamente com a nova palavra. Como o descompressor segue o mesmo algoritmo, mantendo a mesma lista que o compressor, ao ler uma posição, saberá ser ela relativa a uma palavra já existente na lista ou não, e neste caso, que logo em seguida deverá vir a nova palavra. Ao final da codificação/decodificação, a palavra deve ser movida para o início da lista.

Apresentamos na figura 3.22 o algoritmo de codificação para o processo des-

Passo	Lista de Palavras	Palavra lida	Código Gerado
1	ϵ	*viva*	1*viva*
2	*viva*	*maria*	2*maria*
3	*maria viva*	*iá*	3*iá*
4	*iá maria viva*	*iá*	1
5	*iá maria viva*	*viva*	3
6	*viva iá maria*	*a*	4*a*
7	*a viva iá maria*	*bahia*	5*bahia*
8	*bahia a viva iá maria*	*iá*	4
9	*iá bahia a viva maria*	*iá*	1
10	*iá bahia a viva maria*	*iá*	1
11	*iá bahia a viva maria*	*iá*	1

Tabela 3.5: Compressão de Elias-Bentley para exemplo 3.14.

crito acima. O algoritmo de decodificação não será apresentado porque basicamente realiza operações inversas às da codificação.

O exemplo que segue ilustra melhor o processo de codificação. Para facilidade de apresentação, iremos ignorar os espaços em branco, e estaremos codificando apenas as palavras formadas pelas letras do alfabeto romano.

Exemplo 3.14 Seja o seguinte exemplo de texto, formado por versos de uma música em língua portuguesa:

Texto fonte: *viva maria iá iá*
viva a bahia iá iá iá iá

A codificação para o texto está expressa na tabela 3.5. Nota-se que inicialmente a lista está vazia e que no decorrer do processo ela vai sendo preenchida com as novas palavras. Ao final do processo, será obtido o seguinte texto compresso.

Texto compresso: 1*viva* 2*maria* 3*iá* 1 3 4*a* 5*bahia* 4 1 1 1.

Uma modificação para o método MTF foi sugerida por Bentley *et alii*. Esta sugestão foi chamada de *movimento-intermitente-para-frente*[9] e baseia-se na seguinte heurística: o movimento de uma palavra é regulado por um parâmetro $\tau \geq 1$. Assim, após a leitura de uma palavra, ao invés de sempre movê-la para o

[9]Em inglês, *intermittent-move-to-front*.

início da lista, efetua-se este movimento em intervalos de tempo τ. Esta variação é recomendada se o custo de rearrumação das palavras na lista é muito grande.

Outra variação conhecida para a manutenção de listas auto-organizáveis é a da *transposição*. Nesta heurística, sempre que uma palavra é lida ela irá trocar de posição com a palavra anterior na lista. Genericamente, para um valor k, a palavra lida será trocada com a palavra que se encontre k posições anteriores na lista. Elias [Eli75] atribui este esquema genérico a Rivest [Riv76].

Bentley *et alii* e Elias fornecem bibliografias que discutem e analisam diversas outras maneiras para a manutenção de sistemas auto-organizáveis. Dentre estas bibliografias, podemos citar [Knu73], [BM85], [ST85].

É importante mencionar que o método de Elias-Bentley coincide com alguns trabalhos anteriormente apresentados. Ryabko em [Rya87] menciona que os principais resultados do artigo de Bentley *et alii* [BSTW86] coincidem com muitos dos resultados de seu artigo [Rya80]. Neste artigo, o autor propõe uma heurística de compressão denominada *pilha de livros*.[10] Os "livros" seriam as palavras, e sempre que um livro fosse "lido" seria recolocado no início da pilha. Horspool e Cormack em [HC87] também comentam que a proposta apresentada por Bentley *et alii* assemelha-se a um trabalho realizado pelos autores [HC83], mas não publicado. Nesse trabalho, são investigadas várias heurísticas para a manutenção das palavras na lista.

Observa-se que no esquema de Elias-Bentley o código relativo a uma palavra corresponde a um mais o número de palavras diferentes que apareceram no texto desde a sua última ocorrência. Elias em [Eli87] propõe a *codificação por intervalo*, cujo código equivale à quantidade total de palavras, iguais ou não, que apareceram no texto desde a sua última ocorrência. Ou seja, enquanto nesta nova proposta o código representa o *intervalo* desde a última ocorrência da palavra, na codificação de Elias-Bentley o código representa o número de palavras distintas naquele intervalo. Para ilustrar esta diferença, fornecemos abaixo a compressão do texto do exemplo anterior.

Texto compresso: 1*viva* 2*maria* 3*ia* 1 4 6*a* 7*bahia* 4 1 1 1.

Pelo que foi exposto, percebe-se que o método por intervalo tende a obter um desempenho sempre abaixo do desempenho do método Elias-Bentley. Entretanto, este método permite uma implementação extremamente simples e eficiente. Isto porque não é necessária a manutenção de uma lista de palavras, mas apenas um registro para cada palavra da sua última ocorrência no texto. Em virtude deste

[10] Em inglês, *book stack*.

1. Seja t um texto formado de N palavras sobre um alfabeto S de símbolos, $t \in S^*$.
2. Seja *ult* um vetor contendo a última ocorrência de w em t.
   $ult(w) := 0$.
3. Para $i := 1$ até N faça
   (a) Obtenha w de t.
   $w := t(i)$.
   (b) Codifique o valor: $i - ult(w)$.
   (c) Se $ult(w) = 0$ então
   - Codifique w.

   (d) $ult(w) := i$.

Figura 3.23: **Algoritmo para a Codificação por Intervalo.**

fato, apresentamos na figura 3.23 o seu algoritmo de codificação.

A estratégia utilizada pelas diversas heurísticas apresentadas é bastante simples, entretanto alguns aspectos precisam ser esclarecidos.

Inicialmente, é necessária a utilização de alguma técnica de análise léxica que possa reconhecer as palavras no texto. Para textos em língua natural, códigos fonte, é possível a separação do texto em palavras bem-definidas, entretanto para textos binários, de códigos executáveis, imagens sintéticas, fica difícil determinar quais seriam as palavras. Bentley *et alii* utilizam dois grupos de palavras para a realização de experimentos: palavras alfanuméricas, formadas por seqüências de letras do alfabeto romano e números, e palavras não-alfanuméricas. Esta estratégia, entretanto, objetiva mais a compressão de textos em língua natural ou código fonte.

Outro aspecto a ser determinado é a forma como os inteiros e as palavras, na geração do código, devem ser representados. Bentley *et alii* [BSTW86] argumentam que o uso de qualquer código recuperável para representar as palavras (símbolos mais o seu comprimento) e os inteiros é suficiente. Assim, estes elementos podem ser representadas por uma codificação fixa de prefixo. Os autores, entretanto, visando uma maior eficiência e para a garantia do desempenho teórico, sugerem a utilização de uma codificação variável para os inteiros. Elias [Eli87]

também observa este fato e complementa que, embora as probabilidades para os símbolos não precisem ser determinadas, torna-se necessário a utilização de uma codificação de prefixo variável para os inteiros que aumente de comprimento à medida que os valores crescem. Um exemplo desse tipo de codificação é a proposta por Elias em [Eli75]. Estes códigos de Elias foram apresentados em detalhes na seção 2.1.1, p. 12. Desejando-se melhorar a compressão obtida, nada impede que um código estatístico, tal como o de Huffman, [Huf52] ou o Aritmético, [WNC87] possam ser utilizados, tanto na codificação dos inteiros como das palavras.

Avaliação

Bentley *et alii*, com a suposição da codificação variável para inteiros (Elias [Eli75]), demonstram que o método movimento-para-frente, obtém um desempenho no máximo duas vezes mais do que o desempenho do algoritmo de Huffman estático. Também é demonstrado que, à medida que o comprimento do texto tende a infinito, o método alcança a entropia da fonte. Elias [Eli87] também demonstra este resultado para a codificação por ocorrência recente.

Quanto às demais heurísticas sabe-se que, no pior caso, a de movimento-para-frente obtém os melhores resultados para a maioria das distribuições [Eli75]. O esquema de Rivest obtém melhores resultados para quando os símbolos ocorrem no texto de maneira independente [Eli75]. Demonstra-se que a de movimento-intermitente-para-frente também atinge a entropia no limite [BSTW86]. A codificação por intervalo, como visto intuitivamente, tem um desempenho inferior à de movimento-para-frente [Eli75].

Verifica-se que o método Elias-Bentley obtém taxas de compressão muito próximas das obtidas pelo método de Huffman. Quando ocorre fortemente a redundância local, existe uma superação [BSTW86] [Eli87]. Um exemplo desta superioridade é observado em textos que apresentam a seguinte estrutura: $a^k b^k ... n^k$, onde k é o número de ocorrências de cada um dos n caracteres no texto. Enquanto o método de Huffman, na sua modalidade estática, considera este como sendo um texto uniforme, representando cada letra por $\log n$ *bits*, o método Elias-Bentley, irá representar o texto por alguns poucos *bits*.

Deve-se, entretanto, efetuar uma comparação mais cuidadosa do método Elias-Bentley com o método de Huffman na sua modalidade dinâmica. O uso deste método, conjuntamente com o conceito de janela proposta por Knuth em [Knu85], ou de parâmetros proposto por Gallager em [Gal78], uma vez que visa o mesmo objetivo, de eliminação da localidade de referência, deve estar propenso a obter um desempenho semelhante ao método de Elias-Bentley. Por este motivo, achamos interessante efetuar uma comparação empírica entre estes dois métodos.

O resultado desta comparação é apresentado no Capítulo 4.

Implementação

O primeiro aspecto prático a destacar no método Elias-Bentley é que a lista de palavras deverá estar limitada no seu tamanho. Isto significa que a lista comportará, num determinado momento, as palavras que mais ocorrem no texto. Como a inserção de novas palavras é sempre feita no final da lista, se a lista estiver cheia a palavra a ser removida será a do seu final. Este fato estabelece que a heurística para a remoção coincide com a de *Menos-Freqüentemente-Utilizada*, em inglês, *Least-Recently-Used* (LRU).

De acordo com o algoritmo 3.22, as seguintes operações são necessárias: primeiramente, deve-se efetuar a busca das palavras na lista (passo 3 (a)). Para tanto, pode ser utilizada uma *função de espalhamento*, o que leva tempo médio $O(1)$, ou utilizada uma estrutura do tipo *trie*, que leva tempo $O(|w|)$, onde $|w|$ é o comprimento da palavra. Entretanto, as operações cruciais para a eficiência deste método são: na codificação, a *determinação da posição* da palavra na lista (passo 3 (b i)); na decodificação, a operação inversa, que deve *identificar a palavra*, sendo fornecida a sua posição. Nos parágrafos subseqüentes, apresentamos estruturas sugeridas para a realização destas duas operações.

Elias [Eli87] menciona uma implementação econômica, que foi proposta por Rivest [Riv76] para o seu esquema genérico de transposição. Nesta implementação, são utilizados apenas dois vetores de $O(|L|)$, onde $|L|$ é o tamanho da lista de palavras. Um dos vetores representa as palavras nas posições (i.é, o seu índice corresponde às posições), o outro representa as posições das palavras (i.é, o seu índice corresponde à palavra). No esquema de transposição o acesso e a atualização destes dois vetores é feita em $O(1)$, entretanto para a heurística de movimento-para-frente, embora o acesso seja feito em $O(1)$, a atualização leva tempo $O(p)$, onde p é a posição da palavra na lista. Se a lista é muito pequena, ou se uma compressão significativa está sendo obtida esta é uma boa solução.

Para uma implementação mais eficiente, estruturas de dados mais complexas devem ser utilizadas. Bentley *et alii* [BSTW86] apresentam uma implementação que se utiliza de uma árvore binária de busca. Como alternativa, sugerem a utilização de uma árvore do tipo *splay*. Foi esta estrutura que utilizamos na implementação do método para a realização dos experimentos do Capítulo 4, e passamos a expô-la.

Um tipo de árvore *splay* é uma árvore binária de busca auto-ajustável, em que, na medida que um acesso é feito a um nodo, ele é promovido, através de rotações, para a raiz. Garante-se que esta operação é realizada em tempo $O(l)$,

onde l é o nível do nodo na árvore. Uma apresentação detalhada sobre esta estrutura é fornecida por Sleator e Tarjan em [ST85].

Na implementação do método, cada nodo da árvore *splay* deve representar uma palavra. A ordem simétrica (*inorder*) na árvore representa as respectivas posições, em ordem crescente, das palavras. A árvore deve suportar o acesso aos nodos de acordo com a sua posição simétrica; para isto ser possível, além dos apontadores pai e filho, cada nodo deverá conter a sua quantidade de nodos descendentes.

Com a estrutura acima, pode-se realizar as operações desejadas. Para a *determinação da palavra*, fornecida a posição, deve-se percorrer a árvore, a partir da raiz, acumulando as posições percorridas de acordo com os descendentes dos nodos, até se chegar ao nodo na posição desejada. Para a *determinação da posição*, fornecida a palavra, deve-se percorrer a árvore, a partir do nodo que representa a palavra, acumulando no caminho a posição estabelecida pelos descendentes dos nodos, até se chegar a raiz. Para a *atualização da posição da palavra* na árvore é preciso que o nodo seja removido da sua posição e depois inserido na primeira posição. Tem-se que cada acesso, remoção ou inserção de um nodo leva tempo $O(l)$. Sleator e Tarjan estabelecem que dado uma árvore *splay* de n nodos, cada uma destas operações é realizada num limite de *tempo amortizado* de $O(\log n)$. Isto significa que se o que importa é o tempo total de execução, o uso de árvores *splay* é tão eficiente quanto árvores balanceadas. Para uma explicação detalhada sobre complexidade amortizada vide [ST85].

Em relação à implementação do método de codificação por intervalo, verificou-se (através do algoritmo 3.23) que a codificação é imediata, bastando para isso, que se mantenha um vetor com a última ocorrência de cada palavra. A decodificação, entretanto, precisa de uma estrutura mais robusta, pois os códigos podem ser qualquer inteiro no intervalo esparso de 1 até N, sendo N o comprimento do texto.

Uma árvore binária de busca, como a árvore *splay*, pode ser utilizada. Os nodos representam as palavras com seus respectivos intervalos. As palavras são inseridas na árvore de acordo com o seu intervalo. Desejando-se uma decodificação imediata, é preciso considerar um intervalo (das palavras sobre o texto) de tamanho máximo. Desta forma, na codificação, sempre que uma palavra abandona o intervalo, a próxima ocorrência da mesma será feita como se fosse uma nova palavra. Uma maneira eficaz de se codificar/decodificar as palavras com esta restrição é mantê-las numa *memória circular*. O trabalho do decodificador será apenas o de ir preenchendo esta memória com as palavras reconhecidas.

Mäkinen em [Mak89] apresenta implementações para a codificação por intervalo e para a codificação por transposição.

3.2.2 Método de Lempel-Ziv-1977

O objetivo de Ziv e Lempel [ZL77] ao propor este método era o de conseguir uma compressão universal, que, sem nenhum conhecimento das características da fonte de dados, exibisse um processo conjunto de aprendizagem destas características e de codificação dos dados. Este processo deveria possibilitar uma compressão teoricamente eficiente. Esta era uma idéia pioneira, visto que, até aquela época, a maioria dos métodos de compressão existentes apresentavam códigos ótimos com base na suposição do conhecimento da fonte [Huf52].

O método funciona de maneira adaptativa e baseia-se na manutenção de uma janela deslizante sobre o texto fonte, contendo os últimos símbolos reconhecidos do texto. O objetivo é identificar cadeias consecutivas de símbolos (fatores) cada vez maiores no texto lido e que se encontram na janela. O código resultante será a posição e o comprimento do fator na janela.

O método obtido exibe uma compressão teórica tão boa quanto a compressão adquirida com a manutenção de um dicionário de fatores fixos, baseado no conhecimento do texto a ser comprimido [ZL77].

Este método é o resultado de um estudo realizado pelos autores em 1976 sobre a complexidade de seqüências finitas [LZ76]. Em verdade, este estudo principiou, além do método proposto em 1977 [ZL77], outro método proposto em 1978 [ZL78], e que será apresentado na próxima seção (3.2.3). Embora estes dois métodos mantenham diferenças, devido à simultaneidade em que foram apresentados, existe uma confusão em se achar que o método de Lempel-Ziv é um só único.

Como a apresentação do método proposto tinha uma natureza fortemente teórica, vários outros trabalhos surgiram com o intuito de fornecer descrições mais simples para o método. Além disso, muitos outros autores, através de variações à idéia original, propuseram novos métodos. Implementações eficientes também foram sugeridas. Dentre as principais variações, destacam-se a implementação sugerida por Rodeh *et alii* [RPE81], o método proposto por Storer e Szymanski [SS82] [Sto88], os estudos realizados por Bell [Bel86] [Bel87] e os métodos recentemente apresentados por Fiala e Greene [FG89].

Descrição

A compressão originalmente proposta [ZL77] baseia-se na substituição de fatores a serem comprimidos por índices a ocorrências anteriores dos mesmos presentes numa memória interna. Esta memória está organizada como uma *janela deslizante* sobre o texto. Ou seja, a memória é uma porção contínua e à medida que os

símbolos vão sendo reconhecidos, eles vão sendo acrescentados ao final da janela, ao mesmo tempo em que os símbolos do início da janela a abandonam.

Para uma maior eficiência, tanto a janela quanto o fator a ser reconhecido estão limitados no seu comprimento. Se N é o comprimento máximo da janela e F é o comprimento máximo para o fator, a janela será formada de duas partes: uma faixa de comprimento $N-F$, que retrata os últimos símbolos do texto que já foram reconhecidos, e outra parte que contém os próximos F símbolos do texto fonte que ainda não foram compressos. Iremos designar a primeira parte por *janela reconhecida* e a segunda por *janela não-codificada*. Abaixo ilustramos esta divisão graficamente.

Janela	
reconhecida	não-codificada
$(N-F)$	(F)

O próximo fator a ser reconhecido no texto original é o maior prefixo da *janela não-codificada* que se inicia na *janela reconhecida*, e que está limitado no comprimento por F. Observa-se que este maior fator pode conter símbolos da segunda janela, mas não pode iniciar-se nela. Esta condição é necessária para ser possível a posterior descompressão do texto.

Os autores ainda sugerem que no início da codificação a janela seja preenchida com símbolos nulos, por exemplo zeros, ou brancos. Desta forma, a ocorrência destes padrões no texto já pode ser identificada.

O código resultante do processo descrito acima será composto de triplas na forma $\langle d,\ l,\ s\rangle$, onde d é o deslocamento do fator para a *janela reconhecida*, l é o comprimento do fator, e s é o próximo símbolo do texto original que não pôde ser reconhecido; o par $\langle d,\ l\rangle$ é designado *apontador*, e o símbolo s é designado *símbolo de extensão*. O uso do símbolo de extensão é necessário para que a codificação seja mantida, mesmo quando nenhum fator da janela não-codificada não é reconhecido. O código deve ser representado por uma codificação de prefixo fixa, de tal forma que d seja representado por $\lceil \log(1+N-F) \rceil$ *bits* e l seja representado por $\lceil \log(1+F) \rceil$ *bits*.

Após cada codificação dos símbolos no texto (l símbolos do fator mais 1 símbolo de extensão) a janela desliza $l+1$ símbolos para a direita. Ou seja, os $l+1$ símbolos codificados são acrescentados ao final da *janela reconhecida* e os próximos $l+1$ símbolos do texto fonte são acrescentados ao final da *janela não-codificada*.

O processo de codificação descrito acima está expresso no algoritmo da figura 3.24. No restante deste trabalho, estaremos designando este algoritmo por LZ77.

1. Seja t um texto fonte formado de símbolos em S, $t \in S^*$.
2. Seja j uma janela de comprimento $N - F$, inicializada com uma cadeia j_0 constante.
3. **Enquanto** $t \neq \epsilon$ **faça**
   (a) Obtenha o maior fator $f \in jt$, tal que:
      i. $t = fst'$, $s \in S$, $f, t' \in S^*$;
      ii. f inicia-se em j;
      iii. $|f| \leq F$.
   (b) Codifique: $< d(f), |f|, s >$, onde
   $d(f)$ é a distância entre os inícios de f em j e em t.
   (c) $j :=$ o sufixo de jfs de comprimento $N - F$.
   (d) $t := t'$.

Figura 3.24: **Algoritmo para a Compressão de Lempel-Ziv-1977** (LZ77).

Vamos ilustrar este processo a partir do seguinte exemplo:

Exemplo 3.15 Considere que se deseja comprimir o seguinte texto:

Texto fonte: *aababbaababbaaa...*

Considere que o maior fator tenha comprimento de 8, e que a janela tenha comprimento muito grande. Os passos realizados para a compressão estão expressos na tabela 3.6. Observe que no passo de número 4 o maior fator ocupa símbolos da janela não-codificada.

O processo de codificação dará origem ao seguinte texto comprimido:

Texto original:	$\underbrace{a}$	$\underbrace{ab}$	$\underbrace{abb}$	$\underbrace{aababbaaa}$
Texto LZ77:	$\langle 0,0,a \rangle$	$\langle 1,1,b \rangle$	$\langle 2,2,b \rangle$	$\langle 6,8,a \rangle$

Para efetuar a recuperação, o descompressor reconstrói a janela seguindo o mesmo caminho que o compressor. A tabela 3.7 ilustra este processo. Observa-se que a descompressão é extremamente simples e imediata, pois após a leitura de um código basta que seja feita a cópia do fator identificado no final da janela.

Passo	Janela		Texto	Maior	Código
	reconhecida	não-codificada	fonte	fator	gerado
1		*aababbaa*	*babbaaa*	ϵ	$\langle 0,0,a\rangle$
2	*a*	*ababbaab*	*abbaaa*	*a*	$\langle 1,1,b\rangle$
3	*aab*	*abbaabab*	*baaa*	*ab*	$\langle 2,2,b\rangle$
4	*aababb*	*aababbaa*	*a*	*aababbaa*	$\langle 6,8,a\rangle$

Tabela 3.6: Compressão LZ77 para o exemplo 3.15.

Passo	Janela	Texto Compresso	Fator
1		$\langle 0,0,a\rangle\langle 1,1,b\rangle\langle 2,2,b\rangle\langle 6,8,a\rangle$	*a*
2	*a*	$\langle 1,1,b\rangle\langle 2,2,b\rangle\langle 6,8,a\rangle$	*ab*
3	*aab*	$\langle 2,2,b\rangle\langle 6,8,a\rangle$	*abb*
4	*aababb*	$\langle 6,8,a\rangle$	*aababbaaa*

Tabela 3.7: Descompressão LZ77 para o exemplo 3.15.

Storer em [Sto88] apresenta um método *on-line*, designado por *dicionário deslizante*,[11] que vem a ser exatamente uma sugestão prática para o método LZ77.

Nos próximos parágrafos, iremos apresentar as principais variações, sugeridas por alguns autores, em relação ao método originalmente proposto em 1977.

Rodeh *et alii* (1981) (LZR) O método LZ77 foi obtido a partir de um estudo sobre a complexidade de textos finitos que os autores realizaram em 1976 [LZ76]. A diferença entre o método que aquele estudo sugere e o método LZ77 está na não necessidade de se limitar o comprimento da janela sobre o texto. Rodeh *et alii* [RPE81] forneceram uma implementação eficiente para a compressão de Lempel-Ziv considerando tanto o uso de "memória limitada" como o uso de "memória ilimitada". Neste segundo caso, como o comprimento do fator e do seu deslocamento na janela $\langle d,\ l\rangle$ são indefinidos, uma codificação de prefixo de comprimento variável precisa ser utilizada para a sua representação. Dentre os códigos variáveis sugeridos encontra-se o código de Elias [Eli75], apresentado na seção 2.1.1, p. 12.

Storer e Szymanski (1982) (LZSS) Storer e Szymanski em [SS82] generalizaram a classe de compressores a que os métodos Lempel-Ziv pertencem. Esta

[11] Em inglês, *sliding dictionary*.

Passo	Janela		Texto fonte	Maior fator	Código gerado
	reconhecida	não-codificada			
1		*aababbaa*	*babbaaa*	ϵ	*a*
2	*a*	*ababbaab*	*abbaaa*	*a*	*a*
3	*aa*	*babbaaba*	*bbaaa*	ϵ	*b*
4	*aab*	*abbaabab*	*baaa*	*ab*	$\langle 2,2\rangle$
5	*aabab*	*baababba*	*aa*	*ba*	$\langle 3,2\rangle$
6	*aababba*	*ababbaaa*	ϵ	*ababbaa*	$\langle 6,7\rangle$
7	*aababbaababbaa*	*a*	ϵ	*a*	*a*

Tabela 3.8: Compressão LZSS para o exemplo 3.15.

classe foi designada por OPM/L (Modelo Macro de Apontador Original Restrito a Apontadores Esquerdos)[12]. Ao mesmo tempo, foi proposta uma otimização em relação à idéia original [ZL77].

A otimização proposta evita a geração do *símbolo de extensão* no código. Acredita-se que, na prática, considerando-se um alfabeto fonte de tamanho limitado e não muito grande, o gasto para estar sempre representando o símbolo de extensão no código é substancial [Sto88]. Além disso, o símbolo de extensão pode, por sua vez, iniciar outro fator no texto.

Para a realização desta modificação, é sugerida a utilização livre de apontadores e de símbolos no texto compresso. Ou seja, o código será formado ou do apontador $\langle d,\ l\rangle$ ou do símbolo s. A geração do símbolo s será feita quando o apontador ocupar mais espaço do que fator que ele representa. Por exemplo, se o apontador ocupa um espaço de 2 símbolos e o fator que ele representa é de apenas um símbolo, então este símbolo deve ser gerado diretamente no texto compresso. Para ser possível a decodificação, um *bit* indicador deve ser acrescentado ao código para diferenciar apontadores de símbolos.

Para uma ilustração deste processo, a tabela 3.8 apresenta a codificação para o exemplo 3.15. Está-se considerando que o apontador ocupa espaço de um símbolo.

Fiala e Greene (1989) (LZFG) Recentemente, Fiala e Greene [FG89] apresentaram um grupo de métodos baseados numa mesma heurística de compressão. Os cinco métodos propostos são o resultado prático de variações e escolha cuidadosa de parâmetros. Alguns desses parâmetros são o comprimento da janela utilizada, ou a codificação de prefixo adotada para representar os códigos. Os métodos

[12]Em inglês, *Original Pointer Macro Restricted to Left Pointers.*

assumem um compromisso ora na quantidade de memória interna utilizada ora no tempo de compressão ora na taxa de compressão.

A variação apresentada em relação ao LZ77 é semelhante à modificação introduzida pelo LZSS. O *símbolo de extensão* não é gerado no código. Em seu lugar, ao invés do símbolo individual ser gerado no texto compresso, tal como no LZSS, é gerado um *literal*, ou seja, uma cadeia de símbolos consecutivos é copiada diretamente no texto compresso. Portanto, o código será formado ou do apontador $\langle d, l \rangle$ ou do código para um literal w, que equivale ao $|w|$ seguido dos seus respectivos símbolos. Este literal, por sua vez, deverá ter um comprimento máximo W. Para um ganho rápido de contexto, os autores acreditam ser a geração de literais uma boa política no início do processo.

A opção entre se gerar apontador ou literal, em cada passo da codificação, obedece à seguinte heurística: tal qual no LZ77, o método inicialmente determina o mais longo fator f na janela não-codificada. Se $|f|$ for suficientemente grande, é codificado o apontador $\langle d(f), |f| \rangle$. De outra maneira, inicia-se a geração de um literal w de comprimento mínimo, através do consumo de símbolos da janela não-codificada; ao mesmo tempo, na leitura de cada símbolo, busca-se um fator f' de comprimento suficientemente grande. Quando f' é encontrado, este processo termina; codifica-se então o $|w|$ seguido dos seu símbolos, e o apontador $\langle d(|f'|), |f'| \rangle$. Se acontecer de f' não ser encontrado até que w atinja seu comprimento máximo W, então somente o código para w é gerado.

Para ser possível a descompressão, é preciso estabelecer uma distinção entre o código relativo ao literal e o código de apontador. Para o primeiro método proposto pelos autores (método A1) sugere-se a seguinte distinção: utiliza-se 8 *bits* para representar o literal e 16 para o apontador. Se os primeiros 4 *bits* são zero então o código é de literal e os próximos 4 *bits* indicarão o comprimento do literal. Seguindo-se a este *byte* deverão vir os símbolos. Para o apontador, os primeiros 4 *bits* estabelecem o comprimento do fator, e os 12 restantes estabelecem o deslocamento na janela. Considera-se neste caso que o fator é suficientemente grande se o seu comprimento é maior que dois.

De acordo com esta heurística, apresentamos abaixo a codificação LZFG-A1 para o exemplo 3.15:

Texto original:	$\underbrace{aababb}$	$\underbrace{aababbaa}$	$\underbrace{a}$
Texto LZFG:	$(6)aababb$	$\langle 6,8 \rangle$	$(1)a$

Representação dos Códigos Uma sugestão fornecida por muitos autores é o uso de uma codificação de prefixo variável, tanto para o apontador como para os símbolos, com o intuito de reduzir ainda mais a compressão. Em alguns

casos, quando os elementos têm comprimento muito grande ou indefinido, esta codificação é essencial.

Alguns autores sugerem o uso de uma codificação estatística, tal como a de Huffman, ou a Aritmética. Este processo equivale a aplicar um método estatístico na saída da compressão Lempel-Ziv. Acredita-se entretanto, que o uso destes códigos ótimos não forneça reduções muito significativas à compressão, isto porque a geração dos apontadores e símbolos tende a ser feita de maneira uniforme. Intuitivamente, isto ocorre porque os próximos fatores a serem codificados tendem a ser prefixos de fatores ainda mais longos na janela reconhecida. Bell *et alii* [BWC89] ressaltam que este processo compromete a simplicidade do método Lempel-Ziv, e na prática, o gasto, tanto de tempo como de memória, para a manutenção destes códigos não compensa o seu uso. Fiala e Greene [FG89], através de experimentos realizados, não recomendam a codificação de Huffman, pois a mesma não fornece reduções muito altas. Storer [Sto88] acredita que, para fontes bastante grandes, uma modelagem de ordem maior que zero dos elementos deve proporcionar uma redução significativa.

Em virtude dos motivos apresentados acima, os autores freqüentemente optam por outras codificações. Rodeh *et alii* [RPE81] e Bell *et alii* [BWC89] sugerem a codificação de Elias [Eli75]; Fiala e Greene [FG89] apresentam e utilizam um código particular.

Avaliação

Na sua apresentação, Lempel e Ziv [ZL77] demonstraram que se a janela tem comprimento suficientemente grande, então o método proposto tem desempenho tão bom quanto o de um método de codificação de fatores, utilizando um dicionário estático, que se baseie no conhecimento do texto a ser compresso.

Além deste bom desempenho teórico, o método tem se constituído, na prática, em um excelente compressor. Isto devido às características mencionadas na seção 3.2. Dentre estas características citamos a simplicidade do método; a possibilidade de se gerar códigos alinhados com a palavra da máquina (o método LZFG-A1, descrito anteriormente, é um exemplo); a rapidez com que a descompressão pode ser realizada, o que o torna um compressor excelente para aplicações que necessitam comprimir apenas uma vez e expandir variadas vezes; exemplos deste tipo de aplicação são os manuais *on-line*, os *hipertextos*, etc.

Observações importantes podem ser extraídas sobre a estratégia adotada pelo método para a determinação do dicionário de dados.

Tem-se que a simples manutenção de uma janela deslizante sobre o texto apresenta alguns desperdícios. Um deles é a manutenção das várias ocorrências de

um mesmo fator na janela; este aspecto tende a ocorrer devido à própria filosofia do método que espera que os próximos fatores reconhecidos sejam os recentemente acrescidos à janela. Estas várias ocorrências ocupam espaço desnecessariamente. Alguns outros gastos são mencionados por Storer em [Sto88].

Outro aspecto importante é o da remoção dos símbolos da janela. A remoção adotada restringe-se a abandonar os símbolos do início da janela. Uma vez mais, este processo acompanha a filosofia do método. Entretanto, estratégias mais elegantes para a manutenção dos fatores mais freqüentes do texto na janela poderiam ser úteis.

Implementação

Quando o método foi proposto, a abordagem utilizada foi muito mais teórica do que prática. Assim, os autores não tiveram por preocupação a sua implementação. Entretanto, estruturas eficientes devem ser utilizadas para a a determinação do maior fator da janela (passo 3 (a) do algoritmo 3.24). Isto porque, uma implementação direta, embora esteja restrita a $(N - F) * F$ comparações a cada passo, onde N é o comprimento máximo para a janela e F é o comprimento máximo para o fator, pode resultar em milhares de comparações.

Alguns autores propuseram algoritmos lineares para a realização desta busca. Rodeh *et alii* [RPE81] e Fiala e Greene [FG89] realizam a manutenção da janela através de modificações à árvore de sufixo sugerida por McCreight [McC76]. Esta árvore descende da árvore PATRICIA [Mor68], que por sua vez descende de uma estrutura do tipo *trie* [Knu73]. Uma árvore de sufixo é a estrutura adequada para quando várias buscas de um mesmo padrão são realizadas num texto fixo. Bell [Bel86], entretanto, acredita que as árvores de sufixo são estruturas muito complexas para a simples manutenção de uma janela. Assim, visando uma maior eficiência de tempo e espaço, propõe o uso de árvores binárias de busca.

Dentre todas as variantes do método LZ77, resolvemos implementar a LZSS com o intuito de realizar os experimentos do Capítulo 4. A implementação utilizou árvores binárias de busca, seguindo a sugestão fornecida por Bell [Bel86]. Fornecemos abaixo as informações relevantes sobre esta implementação.

A janela utilizada está organizada como uma memória circular. Isto significa que se o símbolo ocorrendo no texto fonte na posição j deve ser inserido na janela, a posição que ocupará será j mod N. Desta maneira, o elemento d do apontador, que estabelece o deslocamento do fator no texto, será a própria posição do padrão na janela.

Considere os $N - F$ fatores de comprimento F que a *janela reconhecida* comporta e considere o fator f de comprimento F da *janela não-codificada*. Bell

observou que se esses fatores fossem ordenados lexicograficamente, f situando-se entre dois fatores, digamos f_a e f_b, de tal forma que: $f_a \leq f \leq f_b$, então o maior fator a ser encontrado ou é o prefixo de f em f_a ou é o prefixo de f em f_b.

Tal fato possibilita o uso de uma árvore binária de busca, ordenada lexicograficamente, em que cada nodo representa um fator da *janela reconhecida*. A busca do próximo fator f é feita percorrendo-se o caminho da raiz até o lugar em que este fator será inserido. Os fatores f_a e f_b necessariamente estarão neste caminho e a determinação do comprimento do maior fator é estabelecida durante o percurso.

Como a codificação de cada símbolo no texto fonte origina um novo fator, ao mesmo tempo em que elimina outro, o tempo de codificação para cada símbolo é o tempo para a inserção e remoção do fator na árvore. Se a árvore está balanceada, o tempo para cada inserção é de $O(F * logN)$ comparações de símbolos. Como o fator a ser excluído da árvore já é conhecido, a remoção consiste em efetuar as atualizações dos apontadores.

A árvore é representada num vetor de N posições. O nodo na i-ésima posição identifica o fator naquela posição da janela. Devido a este fato, cada nodo na árvore apenas precisará conter apontadores para nodos filhos e pai. Ao todo, as estruturas requeridas ocupam espaço $O(N)$.

Verificou-se que existe a liberdade de escolha para os valores do comprimento da janela e do fator. Bell [Bel86], baseado em resultados de experimentos práticos, propõe o uso de $N = 2^{13}$ símbolos e de $16 \leq F \leq 32$ símbolos. Fiala e Greene [FG89] acreditam que $N = 2^{12}$ e $F = 16$ já forneça uma compressão substancial. Para possibilitar um desempenho menos sensível à escolha destes parâmetros, Bell [Bel87] ainda fornece uma implementação que aumenta o comprimento da janela na medida em que vai sendo necessário. Assim, no início a janela comporta 2^0 símbolo, posteriormente 2^1 símbolos, e assim sucessivamente. No Capítulo 4, realizamos experimentos com a variação destes parâmetros.

3.2.3 Método de Lempel-Ziv-1978

O método apresentado por Lempel e Ziv em 1978 [ZL78], conforme mencionado na seção anterior (3.2.2), foi lançado paralelamente à apresentação de outro método proposto pelos mesmos autores em 1977 [ZL77]. Enquanto o método LZ77 mantém uma janela contínua sobre o texto fonte, este novo método, que iremos designar por LZ78, mantém um dicionário de fatores fixos. Isto significa que, enquanto no LZ77 há a possibilidade de se referenciar qualquer fator do texto já lido, no LZ78 apenas fatores fixos do dicionário podem ser referenciados; esses fatores são acrescentados ao dicionário sempre com base nos fatores já

reconhecidos no texto.

O método funciona de maneira adaptativa e mantém um dicionário de fatores fixos. O objetivo é reconhecer fatores longos no texto fonte que se encontram no dicionário. Cada entrada do dicionário será formada pelo fator reconhecido no texto, juntamente com o próximo símbolo que não pôde ser reconhecido. O código resultante do processo será o índice do fator no dicionário.

A nova heurística possibilita o uso de uma memória de tamanho ilimitado, e simplifica o processo de codificação, facilitando a busca dos fatores do texto no dicionário. Além deste bom desempenho prático, Hansel em [Han89] recentemente demonstrou um importante resultado teórico, que é a capacidade do método [ZL78] alcançar adaptativamente a entropia de qualquer fonte de dados.

Da mesma forma que no método anterior (LZ77), a abordagem utilizada na apresentação deste método foi fortemente teórica. Este fato possibilitou o surgimento de novos trabalhos, propondo novas variações em relação à idéia original. Dentre os principais trabalhos, destacam-se a implementação sugerida por Welch [Wel84], as sugestões fornecidas por Miller e Wegman [MW85], a generalização e as extensões ao método realizadas por Storer [Sto88].

Descrição

Verificou-se na seção 3.2.2 que o uso de uma janela deslizante sobre o texto para a representação do dicionário oferece alguns desperdícios (por ex., várias ocorrências de um mesmo fator na janela). Este fato, somado à necessidade de uma codificação eficiente para uma memória de tamanho ilimitado, incentivou a apresentação de uma heurística mais elegante para a manutenção dos fatores do texto.

No método apresentado em 1978 [ZL78] é mantido um dicionário de fatores fixos. Em cada codificação é identificado o maior prefixo do texto ainda não-codificado que existe no dicionário como um fator. O código resultante será o índice do fator no dicionário, juntamente com o símbolo do texto que não pôde ser reconhecido. Após este reconhecimento, o fator e o símbolo são acrescidos ao dicionário como uma nova entrada.

Esta nova heurística procura seguir a mesma filosofia de aprendizagem das características do texto do método LZ77. Naquele método, esperava-se que os próximos códigos referenciassem os fatores recentemente reconhecidos. O método LZ78, através da criação de fatores que são formados por fatores recentemente codificados, está mantendo esta filosofia.

O código resultante do processo será composto de duplas na forma: $\langle i,\ s \rangle$, onde i é o *índice* do fator no dicionário e s é o *símbolo de extensão*. Da mesma

Passo	Texto fonte	Maior fator	Dicionário			Código gerado
			i	fator	*prefixo* $+s$	
1	*aababababaaaab*	ϵ	1	a	$0a$	$\langle 0, a\rangle$
2	*abababaaaab*	a	2	ab	$1b$	$\langle 1, b\rangle$
3	*ababaaaab*	ab	3	aba	$2a$	$\langle 2, a\rangle$
4	*baaaab*	ϵ	4	b	$0b$	$\langle 0, b\rangle$
5	*aaaab*	a	5	aa	$1a$	$\langle 1, a\rangle$
6	*aab*	aa	6	aab	$5b$	$\langle 5, b\rangle$

Tabela 3.9: Compressão LZ78 para exemplo 3.16.

maneira que no LZ77, o uso de s é necessário para que a codificação seja mantida, mesmo quando nenhum símbolo é reconhecido no texto. Observa-se que não existe limitação no tamanho do dicionário, que cresce com a sucessiva inclusão de fatores; assim, a codificação para i deve ser variável e o seu comprimento deve acompanhar o crescimento do dicionário. Se o dicionário, num determinado momento, contém $|D|$ fatores, i está sendo representado por $\lceil \log(1 + |D|) \rceil$ *bits*.

Uma característica importante deste processo é que cada entrada do dicionário consiste de um fator que também é uma entrada do dicionário, seguido de um símbolo de extensão. Ou seja, todos os prefixos de uma entrada estão presentes no dicionário. Este fato possibilita uma representação eficiente do dicionário, em que cada entrada tem um comprimento fixo, formado pelo índice do fator mais o símbolo de extensão.

Vamos ilustrar este processo a partir do seguinte exemplo:

Exemplo 3.16 Considere que se deseja comprimir o seguinte texto:

Texto fonte: *aababababaaaab...*

Os passos realizados para a codificação e para a construção dinâmica do dicionário de dados estão expressos na tabela 3.9. Em cada passo é descrita a entrada do dicionário, gerada após o reconhecimento do maior fator no texto. Observe que o campo *prefixo* $+s$ fornece a representação fixa para cada fator do dicionário; *prefixo* é o índice do fator e s é o símbolo de extensão.

O processo de codificação dará origem ao seguinte texto comprimido:

$$\begin{array}{lcccccc} \textit{Texto original}: & \underbrace{a} & \underbrace{ab} & \underbrace{aba} & \underbrace{b} & \underbrace{aa} & \underbrace{aab} \\ \textit{Texto}\ \text{LZ78}: & \langle 0, a\rangle & \langle 1, b\rangle & \langle 2, a\rangle & \langle 0, b\rangle & \langle 1, a\rangle & \langle 5, b\rangle \end{array}$$

Passo	Texto compresso	Fator gerado	Dicionário		
			i	fator	*prefixo* $+s$
1	$\langle 0,a\rangle\langle 1,b\rangle\langle 2,a\rangle\langle 0,b\rangle\langle 1,a\rangle\langle 5,b\rangle$	a	1	a	$0a$
2	$\langle 1,b\rangle\langle 2,a\rangle\langle 0,b\rangle\langle 1,a\rangle\langle 5,b\rangle$	ab	2	ab	$1b$
3	$\langle 2,a\rangle\langle 0,b\rangle\langle 1,a\rangle\langle 5,b\rangle$	aba	3	aba	$2a$
4	$\langle 0,b\rangle\langle 1,a\rangle\langle 5,b\rangle$	b	4	b	$0b$
5	$\langle 1,a\rangle\langle 5,b\rangle$	aa	5	aa	$1a$
6	$\langle 5,b\rangle$	aab	6	aab	$5b$

Tabela 3.10: Descompressão LZ78 para exemplo 3.16.

Para efetuar a recuperação, o descompressor reconstrói o dicionário seguindo o mesmo caminho que o compressor. Ao ler o índice, deverá percorrer o dicionário de acordo com o prefixo das entradas, gerando o fator em ordem reversa. A tabela 3.10 ilustra este processo.

O processo de codificação apresentado está expresso no algoritmo da figura 3.25.

A seguir, iremos apresentar as principais variações ao método LZ78, fornecidas por alguns autores.

Welch (1984) (LZW) Welch em [Wel84] apresenta uma variação ao método LZ78 que é equivalente à modificação apresentada pelo LZSS ao LZ77. Ou seja, a geração do *símbolo de extensão* é evitada. O código resultante será composto apenas de índices. Conforme será visto, para tornar a decodificação possível o dicionário deve ser inicializado com o alfabeto de símbolos.

Na codificação, os fatores são acrescidos ao dicionário da seguinte forma: após a identificação do maior fator do texto no dicionário, o índice deste fator é codificado; ao mesmo tempo, este índice mais o símbolo que não pôde ser reconhecido geram uma nova entrada no dicionário. Para uma ilustração deste processo, a tabela 3.11 apresenta a compressão para o exemplo 3.16.

Na decodificação, decodifica-se uma fator f a partir do seu índice i; o fator f', reconhecido anteriormente, seguido do primeiro símbolo de f geram uma nova entrada no dicionário.

O processo de codificação descrito faz com que o LZW esteja sempre acrescentando um fator no dicionário que ainda não ocorreu no texto. De tal maneira que o descompressor está sempre um passo atrasado em relação ao compressor. Este fato pode causar transtorno à descompressão. A tabela 3.12, que apresenta a descompressão para o exemplo 3.16, ilustra este problema. No passo 5 o descom-

1. Seja t um texto fonte formado de símbolos s em S, $t \in S^*$.
2. Seja D um dicionário.
   $D := \{\epsilon\}$.
3. **Enquanto** $t \neq \epsilon$ **faça**
   (a) Obtenha o maior fator $f \in D$, tal que:
   $t = fst'$, $s \in S$, $f, t' \in S^*$.
   (b) Codifique: $< i(f), s >$, onde $i(f)$ é o índice de f em D.
   (c) Acrescente fs a D.
   $D := D + fs$.
   (d) $t := t'$.

Figura 3.25: **Algoritmo para a Compressão de Lempel-Ziv-1978** (LZ78).

Passo	Texto fonte	Maior fator	Dicionário			Código gerado
			i	fator	*prefixo* $+s$	
			1	a	$0a$	
			2	b	$0b$	
1	$aababababaaaab$	a	3	aa	$1a$	1
2	$abababaaaab$	a	4	ab	$1b$	1
3	$bababaaaab$	b	5	ba	$2a$	2
4	$ababaaaab$	ab	6	aba	$4a$	4
5	$abaaaab$	aba	7	$abaa$	$6a$	6
6	$aaab$	aa	8	aaa	$3a$	3
7	ab	ab	–	–	–	4

Tabela 3.11: Compressão LZW para exemplo 3.16.

Passo	Texto compresso	Fator gerado	Dicionário		
			i	fator	*prefixo* $+s$
			1	a	$0a$
			2	b	$0b$
1	1 1 2 4 6 3 4	a	–	–	–
2	1 2 4 6 3 4	a	3	aa	$1a$
3	2 4 6 3 4	b	4	ab	$1b$
4	4 6 3 4	ab	5	ba	$2a$
5	6 3 4	aba	6	aba	$4a$
6	3 4	aa	7	$abaa$	$6a$
7	4	ab	8	aaa	$3a$

Tabela 3.12: Descompressão LZW para exemplo 3.16.

pressor lê o índice 6, sem ainda ter esta entrada no dicionário. Genericamente, este problema ocorre quando o texto lido apresenta a seqüência $swsws$, e sw já ocorreu no texto, mas sws ainda não. Ao ler a seqüência, o compressor codifica sw e acrescenta sws na tabela, digamos com índice i; em seguida codifica sws com o referido índice i. O descompressor, ao reconhecer sw ainda não acrescentou na tabela o fator sws; ao ler o próximo código, a saber, o índice i de sws, verifica que equivale a uma entrada ainda não gerada. Entretanto, ele sabe que esta nova entrada será formada do fator sw seguido do primeiro símbolo desta nova entrada; portanto, este primeiro símbolo necessariamente é s, e a nova entrada será sws.

Storer (1988) (LZS) Storer em [Sto88] apresenta uma generalização para o LZ78, visando principalmente uma adequação prática para o método. Esta generalização originou um conjunto de métodos designados por *dicionário dinâmico*[13].

Inicialmente, aconselha-se evitar a geração do *símbolo de extensão*. Portanto, o dicionário de fatores deve ser inicializado com os símbolos do alfabeto, tal qual o LZW.

Storer observou que na heurística adotada pelo LZ78 uma entrada para o dicionário é formada por um fator já reconhecido, conjuntamente com um símbolo, que seria o primeiro símbolo do próximo fator a ser reconhecido no texto. O método LZW, por exemplo, realiza exatamente este processo para a geração das suas entradas. Em outras palavras, se p foi o último fator reconhecido e $c = sc'$ é o fator atual, então a próxima entrada do dicionário será ps. Esta idéia, de

[13] Em inglês, *dynamic dictionary*.

gerar a entrada de acordo com o o último fator e com os símbolos do fator atual, pode ser estendida de maneira a possibilitar diferentes heurísticas de atualização do dicionário.

Storer apresenta três heurísticas de atualização. Estas heurísticas estão listadas a seguir:

- Primeiro caractere (PC)[14] — é a anteriormente descrita e a adotada pelo LZ78. Considera-se como entrada a concatenação de p com o primeiro símbolo de c, ou seja, ps.
- Identidade (ID)[15] — considera-se como entrada a concatenação de p com c, ou seja, pc. Com esta estratégia, nem todos os prefixos de um fator estarão presentes no dicionário.
- Todos os prefixos (TP)[16] — considera-se como entrada do dicionário o fator p concatenado, não apenas com o primeiro símbolo de c, mas com todos os prefixos de c. Por exemplo, se $p = a$ e $c = abb$, as entradas geradas são aa, aab e $aabb$. Conforme se verifica, esta heurística inclui os fatores das heurísticas anteriormente descritas.

Para ilustrar esta idéia, apresentamos na figura 3.13 a compressão para o exemplo 3.16 utilizando-se da estratégia ID. Observe que a descompressão de qualquer heurística do LZS não terá os problemas do LZW, pois uma entrada é acrescida ao dicionário sempre com base no fator anteriormente reconhecido.

Storer também observa que embora o dicionário possa crescer indefinidamente, na prática, o seu tamanho deve ser limitado. Este fato determina a utilização de heurísticas para a remoção dos fatores do dicionário.

A heurística de remoção mais imediata é a FREEZE, em que excedendo-se o tamanho do dicionário, não serão mais acrescidos fatores a ele. Outra heurística simples é a CLEAR, em que havendo estouro, todas as entradas são limpas, e o compressor reinicia a aprendizagem do contexto. Uma heurística mais sofisticada é a LRU (*Least Recentley Used*), *Menos Recentemente Utilizada*, onde os fatores removidos são aqueles menos utilizados. Além destas, uma grande variedade de heurísticas pode ser imaginada.

Através da combinação destas heurísticas, de atualização e remoção de fatores no dicionário, uma enorme quantidade de métodos pode ser extraída. A maioria dos métodos propostos por variados autores se adequam a uma destas estratégias.

[14] Do inglês *first character*.
[15] Do inglês *identity*.
[16] Do inglês *all-prefixes*.

Passo	Texto fonte	Maior fator	Dicionário		Código gerado
			i	fator	
			1	a	
			2	b	
1	$aababababaaaab$	a	–	–	1
2	$abababaaaab$	a	3	aa	1
3	$bababaaaab$	b	4	ab	2
4	$ababaaaab$	ab	5	bab	4
5	$abaaaab$	ab	6	$abab$	4
6	$aaab$	aa	7	$abaa$	3
7	ab	ab	8	$aaab$	4

Tabela 3.13: Compressão LZS-ID para exemplo 3.16.

Por exemplo, o LZW atualiza o dicionário com base na FC e remove os fatores com base na FREEZE.

Miller e Wegman (1985) (LZMW) As diversas variações propostas por Miller e Wegman [MW85] exibem alguma das heurísticas apresentadas acima.

Numa primeira variação, os autores sugerem que o dicionário seja inicializado com o alfabeto fonte; esta variação é semelhante ao LZW. Numa segunda alternativa, sugere-se a remoção dos fatores do dicionário através da heurística LRU. Numa terceira variação, recomenda-se o uso da heurística ID (identidade) de atualização dos fatores.

Representação dos Códigos Da mesma maneira que no LZ77, para possibilitar uma maior compressão, pode-se pensar em aplicar uma codificação estatística, tal como a de Huffman, ou a Aritmética para representar os índices do dicionário. Storer [Sto88] salienta que possivelmente esta prática não irá proporcionar compressão alguma, pois, pela própria heurística de atualização, os índices tendem a uma distribuição uniforme. Isto acontece porque os fatores que mais ocorrem no texto originam novas entradas tendo os fatores como prefixos; a partir de então estas novas entradas serão mais prováveis do que os fatores que as geraram.

Avaliação

Hansel em [Han89] demonstrou que o método LZ78 tem a capacidade de alcançar adaptativamente a entropia de qualquer fonte de dados. Ou seja, à medida que

o comprimento do texto tende a infinito, a compressão é assintoticamente *ótima*. Este vem a ser um excelente resultado. Entretanto, a compressão só é ótima se o comprimento dos textos é bastante grande, e freqüentemente os textos têm comprimento curto. Além disso, mesmo dispondo de um texto suficientemente grande, não se dispõe de memória de comprimento ilimitado para poder alcançar o resultado ótimo. Bell *et alii* em [BWC89] observam estes aspectos e salientam que o principal ponto a ser obtido é quão rapidamente o método converge para o seu limite. Conclui-se com o fato de que, na prática, esta convergência é relativamente lenta, e o desempenho do LZ78 é comparável ao LZ77.

Além deste excelente desempenho teórico, devido às características mencionadas na seção 3.2, e principalmente por proporcionar implementações altamente eficientes, o método LZ78 tem se constituído, na prática, em um excelente compressor.

Sobre as diversas heurísticas mencionadas por Storer [Sto88], e adotadas por implementações práticas, algumas observações podem ser extraídas.

Acredita-se que o uso das heurísticas de remoção possibilitam a obtenção de uma mesma compressão com uma quantidade menor de memória.

As heurísticas de atualização TP (todos os prefixos) e ID (identidade) possibilitam uma aprendizagem mais rápida das características do texto; em compensação, preenchem mais rapidamente o dicionário de fatores, consumindo mais memória.

Storer realizou experimentos empíricos com estas heurísticas e obteve as seguintes conclusões: a heurística FREEZE não é estável, significando que quando o dicionário fica pequeno, o desempenho decai drasticamente; para as atualizações, o desempenho obtido pela ID-LRU ficou acima da AP-LRU, que por sua vez foi melhor que a FC-LRU.

Implementação

A estrutura natural para a representação do dicionário de fatores do LZ78 é uma árvore do tipo *trie*. Numa árvore *trie*, as arestas são rotuladas por símbolos do alfabeto, e os nodos são marcados como representando uma cadeia de símbolos, ou não.

Para a representação do dicionário de fatores, basta que as arestas da *trie* identifiquem os símbolos do alfabeto, e que os nodos internos identifiquem o índice associado ao fator. Para uma ilustração desta estrutura, a figura 3.26 fornece uma representação para o dicionário da figura 3.9 construído pelo LZ78 para o exemplo 3.16.

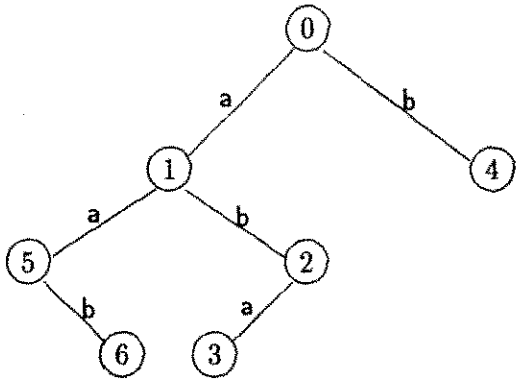


Figura 3.26: *Trie* para o dicionário LZ78 do exemplo 3.16.

Com esta estrutura, a identificação do mais longo fator (passo 3 (a) do algoritmo 3.25) será feita percorrendo-se, a partir da raiz, as arestas que os símbolos do fator indicam, até que não seja mais possível reconhecer símbolos do fator. A inserção do novo fator (passo 3 (c)) será feita pelo acréscimo de um novo nodo na *trie*, que descende do último nodo do fator reconhecido, e deve ser rotulado pelo símbolo de extensão. A decodificação é realizada de maneira inversa, percorrendo-se o caminho do nodo que representa o índice até a raiz.

Iremos a seguir detalhar a implementação utilizada nos experimentos do Capítulo 4, e que se baseou no método LZS (Storer).

Adotamos uma estrutura do tipo *trie*. Diferentemente da estrutura apresentada na figura 3.26, que contém os símbolos associados às arestas, utilizamos uma *trie* cujos símbolos estão representados nos próprios nodos. Além dos símbolos, os nodos conterão o apontador para o nodo pai, e o valor de código associado ao nodo. Esta estrutura possibilita uma decodificação imediata. Na codificação, para a identificação dos nodos na árvore, utilizamos uma *função de espalhamento*, em que fornecido o nodo e o símbolo do alfabeto, é retornado o nodo relativo ao correspondente filho. O uso da *função de espalhamento* admite um tempo constante de acesso médio, além de economizar espaço. Com esta estrutura, o reconhecimento de cada fator pôde ser feito em tempo $O(|f|)$, onde $|f|$ é o comprimento do fator.

Para a atualização dos fatores no dicionário, adotamos a heurística TP (todos os prefixos). Neste caso, ao invés de acrescentar apenas um nodo no final do último fator reconhecido, devem ser acrescidos nodos relativos a todos os símbolos do prefixo do fator atual; cada um desses nodos descenderá do outro recentemente

acrescido. Cuidados devem ser tomados para não realizar o acréscimmo de nodos já existentes.

Adotamos a heurística de remoção LRU. Para tanto, utilizamos, conforme sugerido, uma lista duplamente encadeada. Esta lista contém todos os nodos da árvore e mantém a heurística LRU; assim, sempre que é feito um acesso ao nodo na árvore, este nodo é promovido para o início da lista. Uma vez que na codificação o acesso aos nodos do fator é feito na ordem natural dos símbolos, cuidados devem ser tomados para que os símbolos do ínicio do fator estejam à frente dos demais símbolos do fator na lista. Quando o dicionário estiver cheio, os nodos do final da lista são removidos. A manutenção da lista é feita em tempo constante.

3.3 Compressores Genéricos

Nesta última década, uma grande variedade de utilitários para a compressão genérica de dados tornou-se bastante popular nos diversos ambientes de computação. Este fato deveu-se principalmente à rapidez e à significativa redução proporcionada nos comprimentos dos arquivos por estes sistemas. Assim, estas ferramentas tornaram-se muito úteis para a transferência de arquivos em redes e para o gerenciamento dos arquivos em discos.

A compressão realizada por estes sistemas baseia-se em um ou outro método de compressão anteriormente abordado. Por este motivo, escolhemos apresentá-los nesta seção, e passaremos a designá-los por *compressores genéricos*. Alguns destes compressores são comerciais, outros são de livre distribuição. Dentre estes, escolhemos para abordar os que se destacam, seja pela sua popularidade, seja por proporcionarem uma significativa compressão, seja por adotarem métodos conhecidos. A lista apresentada não é exaustiva, até porque novos compressores e novas versões têm sido continuamente lançados.

Os compressores mais populares fundamentam-se principalmente no método de Lempel-Ziv [ZL77], com as variações propostas por Storer e Szymanski [SS82] (LZSS) e Fiala e Greene [FG89] (LZFG); e no método Lempel-Ziv [ZL78], com a variação proposta por Welch [Wel84] (LZW). Alguns compressores, antes de aplicarem estes métodos, realizam uma compressão que freqüentemente é a codificação por carreiras, do inglês *Run Length Encoding* (RLE). Outros, sobre o texto compresso resultante dos métodos, realizam alguma compressão estatística, utilizando-se principalmente do método de Huffman [Huf52] [Gal78] [Knu85] e do método de Shannon-Fano [SW49].

A apresentação aqui descrita ora se baseia nos fontes desses compressores,

quando disponíveis, ora se baseia na documentação fornecida conjuntamente com os códigos executáveis.

3.3.1 *pack*

O utilitário *pack* está presente no ambiente UNIX e se baseia no método de Huffman estático [Huf52].

O algoritmo utilizado considera como alfabeto fonte os caracteres de representação da máquina e fundamenta-se no cálculo individual das freqüências dos símbolos, realizando portanto uma modelagem independente.

3.3.2 *compact*

O utilitário *compact* (1984) está presente no ambiente UNIX e tem por base o método de Huffman dinâmico [Huf52] [Gal78]. Possivelmente devido ao concorrente *compress* as novas versões do UNIX anunciam que o compressor será desativado em breve.

O algoritmo considera como alfabeto fonte os caracteres de representação da máquina e realiza uma modelagem independente.

3.3.3 *compress*

O programa *compress* (1984) foi desenvolvido pelo grupo da *Free Software Foundation* [TMD+84], que dispôs o seu fonte para livre distribuição. Foi desenvolvido para o sistema UNIX, onde se popularizou. Tem por base o método de Welch [Wel84] (LZW).

O algoritmo utiliza um dicionário dinâmico com códigos de comprimento variável. O código inicia-se em 9 *bits*, podendo ir até 16. Uma limpeza total é realizada quando o dicionário está cheio, mas somente quando a taxa de compressão decresce.

Uma aspecto interessante deste algoritmo é o uso de uma eficiente função de *hashing* para a identificação do fator no dicionário. No LZW cada entrada do dicionário tem comprimento fixo, formada pelo índice do prefixo do fator mais o símbolo de extensão. Na determinação do maior fator, para cada novo símbolo lido a função é aplicada na combinação deste símbolo com o índice do fator até então reconhecido; se a função origina uma entrada já existente, este processo continua, de outra maneira foi identificado o maior fator. É utilizado um endereçamento aberto com duplo *hashing*,[17] baseado no algoritmo D proposto por

[17] Em inglês, *open addressing with double hashing.*

Knuth em [Knu73], (seção 6.4, p. 521); para a primeira exploração é aplicado um ou-exclusivo entre os elementos, e para a segunda exploração é adotada a sugestão de primos relativos de G. Knott.

3.3.4 PKPAK

O utilitário PKPAK (1988) [Kat86] é um sistema de livre distribuição, exceto para instituições de cunho comercial; seus fontes não estão disponíveis. É utilizado principalmente no ambiente DOS. Apresenta várias maneiras de se efetuar a compressão, que são o resultado da utilização conjunta de alguns métodos, a saber: método de Welch [Wel84] (LZW), método de codificação por carreiras, possivelmente o RLE, e método de Huffman estático [Huf52].

O PKPAK escolhe entre quatro modalidades para a compressão dos arquivos. Estas modalidades utilizam as seguintes designações: *Packed, Squeezed, Crunched* e *Squashed.* Não encontramos documentação relativa ao critério de seleção destas modalidades. A seguir, citamos os métodos adotados por estas modalidades.

- *Packed* — Aplica o método de codificação por carreiras (RLE).
- *Squeezed* — Utiliza o método de Huffman estático [Huf52].
- *Crunched* — Tem por base o LZW, e em algumas das três variações utiliza o RLE. As variações são:
  1. Não utiliza o RLE. Trabalha com um dicionário de tamanho estático, cujo código é de 12 bits.
  2. Utiliza o RLE. Trabalha com um dicionário de tamanho estático, cujo código é de 12 bits.
  3. Utiliza o RLE. Trabalha com um dicionário de tamanho dinâmico, cujo código varia de 9 a 12 bits. Realiza uma limpeza total quando o dicionário está cheio, mas somente quando a taxa de compressão decresce.
- *Squashed* — Baseia-se no LZW. Não utiliza o RLE. Trabalha com um dicionário de tamanho dinâmico, cujo código varia de 9 a 13 bits. Realiza uma limpeza total quando o dicionário está cheio, mas somente quando a taxa de compressão decresce.

3.3.5 PKZIP

O PKZIP (1989) [Kat89] é o sucessor do PKPAK, sendo distribuído pela mesma empresa. Tornou-se bastante popular no ambiente DOS, principalmente na utilização para transferência de arquivos em redes. Apresenta várias formas de compressão. Estas diversas formas são o resultado da utilização conjunta de alguns métodos de compressão, que são: método de Welch [Wel84] (LZW), método de Storer e Szymasnki [SS82] (LZSS) método de Shannon-Fano [SW49] [Fan49].

O PKZIP escolhe entre três modalidades de compressão, que têm as seguintes designações: *Shrunk, Reduced* e *Imploded.* Não encontramos documentação relativa ao critério de seleção destas modalidades. Cada uma dessas modalidades é o resultado da utilização conjunta de vários métodos de compressão. A seguir, iremos apresentá-las.

- *Shrunk* — Baseia-se no LZW. O dicionário LZW tem tamanho dinâmico, cujo código varia de 9 a 13 *bits.* A estratégia para a atualização do dicionário é a seguinte: o tamanho de código só é aumentado se as entradas relativas ao novo código são usadas posteriormente para novas codificações. Além disso, quando o dicionário está cheio, ao invés de uma limpeza total é utilizada uma limpeza parcial, em que apenas as folhas da *Trie* são removidas para reutilização.

- *Reduced* — Resultado da utilização conjunta de dois algoritmos particulares. O primeiro, utiliza uma codificação semelhante ao LZFG; o segundo parece aplicar uma compressão estatística sobre os campos de deslocamento e comprimento, resultantes do primeiro algoritmo.

- *Imploded* — Resultado da utilização conjunta do método LZSS e do método de Shannon-Fano (SF). A janela adotada tem comprimento de 4K ou 8K, e o máximo comprimento do fator é de 64. A compressão SF é aplicada sobre os elementos resultantes da compressão LZSS; estes elementos são a posição, comprimento e opcionalmente o símbolo de extensão; no caso da posição, o SF é aplicado somente aos 6 *bits* mais significativos, sendo os menos significativos gerados textualmente. As árvores de Shannon-Fano, por sua vez, são armazenadas no texto comprimido num formato compresso.

3.3.6 ICE

O compressor ICE (1989) [Yos88] é o resultado do esforço conjunto de japoneses *hobbystas.* É um compressor de livre distribuição cujos fontes estão disponíveis.

Seu algoritmo baseia-se na utilização conjunta de dois métodos de compressão, o método de Storer-Szymanski [SS82] (LZSS), com a implementação sugerida por Bell [Bel86], e o método de Huffman dinâmico (HUFDIN) [Huf52] [Gal78].

O LZSS é aplicado ao texto fonte e sobre o seu resultado, que é a posição e o comprimento do fator, juntamente com os caracteres individuais, é aplicado o método HUFDIN. A codificação estatística da posição é apenas aplicada nos 6 *bits* mais significativos, sendo que a mesma ocupa 12 *bits*. Os autores mencionam que também foi construído um algoritmo que utiliza o método Aritmético, com a implementação proposta por Witten *et alii* [WNC87], em substituição ao de Huffman. Verificaram que a sua utilização oferece melhores taxas de compressão, entretanto não conseguiram vencer o tempo gasto por este método, principalmente na descompressão.

3.3.7 LHA

O compressor LHA (1991) [Yos91] é o sucessor do ICE, sendo também de livre distribuição. Os autores substituíram o LZSS pelo método de Fiala e Greene [FG89] (LZFG), e substituíram o Huffman dinâmico pelo Huffman estático. A justificativa apresentada para a mudança é que o LZFG proporciona melhoras significativas na compressão, e o Huffman estático é mais rápido que o dinâmico.

3.3.8 ARJ

O compressor ARJ (1991) [Jun90] é um compressor recente que advoga, na sua documentação, alcançar taxas de compressão superiores a qualquer outro utilitário até então conhecido. É um sistema de livre distribuição, exceto para instituições de cunho comercial, seus fontes não estão disponíveis. A documentação sobre os métodos adotados é escassa, sendo apenas mencionado o uso conjunto do método Lempel-Ziv-1977 (LZ77) com a codificação de Huffman estática. Existem quatro modalidades de compressão, que assumem um compromisso de tempo e compressão. As três primeiras utilizam os métodos acima, a quarta não utiliza a codificação de Huffman, sendo a mais rápida.

Capítulo 4

Comparação Empírica entre os Métodos

Este Capítulo objetiva realizar uma avaliação de desempenho, a nível de taxa de compressão, dos diversos métodos e compressores genéricos, abordados nas seções anteriores. A análise será feita com base nos experimentos realizados com os métodos implementados e que levou em consideração tipos e comprimentos dos arquivos. Além de uma comparação global, será feita uma avaliação individual para cada método.

4.1 Introdução

Comparar diversos métodos de compressão de dados é uma tarefa árdua. Além da escolha cuidadosa dos elementos em que a comparação se baseará, quais sejam, tipos de arquivos, quantidade e comprimento dos mesmos, some-se a liberdade de variação de parâmetros que alguns dos métodos apresentam. Considerando-se a combinação destes aspectos, o conjunto de resultados obtidos vem a ser de grandeza exponencial.

Nossa avaliação visou a comparação dos métodos e compressores genéricos, a nível de capacidade de compressão alcançada. Considerando-se um conjunto de resultados onde muitos fatores de medição podem ser extraídos, tentou-se, na medida do possível, levá-los em consideração. Como conclusão da avaliação, colheu-se para cada método um conjunto de observações importantes sobre o seu desempenho, a partir da variação dos seus parâmetros. Finalmente, com base nos seus melhores resultados, realizou-se uma comparação entre os diversos métodos e compressores genéricos.

Todos os métodos expostos anteriormente foram implementados, tanto o compressor quanto seu respectivo descompressor, sendo que este foi utilizado numa fase inicial para a verificação da correção dos métodos. Entretanto, para a realização dos experimentos, devido ao grande número de variações e de dados, efetuou-se apenas a compressão. Para a implementação, utilizou-se um ambiente SUN[1] de estações de trabalho. Este ambiente, além de nos fornecer um conjunto extenso de dados, nos permitiu variar livremente a quantidade de memória interna utilizada pelos métodos. Os compressores genéricos *pack*, *compact* e *compress* também foram executados nas estações SUN, enquanto que os demais foram executados num microcomputador PC-XT (640 KB), sob o MS-DOS.[2]

Vale ressaltar que o aspecto da quantidade de memória interna utilizada pelos métodos, assim como o aspecto do tempo de execução são de importância fundamental, pois a depender do ambiente em que os métodos são utilizados e da aplicação a que se destinam, eles podem tornar o método inviável ou muito competitivo em relação aos demais. Entretanto, embora estes aspectos merecessem destaque, não foi nosso objetivo avaliá-los, de tal forma que a implementação realizada não os levou em consideração. Seria necessário mencionar que, em parte, esta decisão deveu-se principalmente a grande carga de trabalho dedicada para a consecução da implementação e dos experimentos.

Alguns dos autores, na medida em que expuseram seus métodos, procuraram

[1] SUN é uma marca registrada da *SUN Microsystems*, Inc.

[2] MS-DOS é uma marca registrada da *Microsoft*, Inc.

efetuar uma comparação empírica entre suas várias modalidades e alguns outros métodos. Este é o caso de Knuth em [Knu85], quando realiza experimentos para o FGK, com e sem janelas; Witten *et alii* em [WNC87], que comparam o método aritmético com o Huffman adaptativo, através do utilitário *compact*, a nível de tempo e taxa de compressão; Bentley *et alii* em [BSTW86], que comparam o método MTF com o Huffman adaptativo sem janelas, considerando caracteres e palavras; Bell em [Bel86], que compara alguns dos métodos Lempel-Ziv com os métodos aritmético e Huffman adaptativo, a nível de tempo, quantidade de memória e taxa de compressão; Storer em [Sto88], que compara os vários métodos que propõe, a saber, *dicionário estático, deslizante e dinâmico*, analisando a variação de alguns parâmetros. Comparações mais abrangentes, considerando uma maior quantidade de dados e de métodos, eficiência de compressão e tempo de execução, e quantidade de memória interna utilizada, foram feitas por Bell *et alii* em [BWC89], que comparam vários compressores aritméticos com vários compressores Lempel-Ziv, e por Fiala e Greene em [FG89], que comparam os cinco métodos que propõem com o FGK [Knu85], Vitter [Vit87], Cleary e Witten [CW84], Miller e Wegman [MW85], *compress* [TMD+84] e Bentley *et alii* [BSTW86].

Nosso experimento considera uma quantidade razoável de métodos e dados; também incorpora algumas hipóteses mencionadas pelos autores mas não implementadas, tais como a codificação estatística na saída dos compressores Lempel-Ziv, ou a comparação entre o método proposto por Bentley *et alii* [BSTW86] e o método de Huffman dinâmico com janela, proposto por Knuth [Knu85].

4.2 Elementos da Avaliação

4.2.1 Dados

Escolheu-se quatro tipos de dados para a realização dos experimentos. Estes dados estão enumerados abaixo.

1. DOC — Este conjunto de dados é formado de textos em linguagem natural. Foram considerados arquivos nos formatos LaTeX, escritos em língua portuguesa e inglesa por diversos autores, e arquivos no formato *troff*[3], contendo as páginas de manuais[4] em inglês dos utilitários presentes no ambiente SUN.

[3] *troff* é um utilitário do sistema UNIX.

[4] Em inglês, *man pages*.

DADOS							
Comprimento			Tipo				
			DOC	PROG	EXEC	BITM	Total
PEQUENO	#		11	71	6	1	89
(< 10 KB)	total	(KB)	60	289	20	9	378
	média	(KB)	5	4	3	9	4
MÉDIO	#		27	43	18	10	98
(< 100 KB)	total	(MB)	1	1,2	1	0,6	3,8
	média	(KB)	38	29	57	61	40
GRANDE	#		2	1	10	7	20
(> 100 KB)	total	(MB)	0,8	0,1	3,2	1,3	5,4
	média	(KB)	410	102	328	190	276
Total	#		40	115	34	18	**207**
	total	(MB)	1,9	1,6	4,2	1,9	**9,6**
	média	(KB)	49	14	127	108	**47**

Tabela 4.1: Quantidade e comprimento de cada tipo de dado por categoria de comprimento.

2. PROG — Este conjunto é formado por programas fontes nas linguagens C, C++, PASCAL e LISP. No caso das três primeiras linguagens, os programas foram obtidos de vários programadores; no caso de LISP, a amostra foi retirada do conjunto de fontes para o *GNU-Emacs.*[5]

3. EXEC — Este conjunto considera arquivos executáveis de utilitários presentes no sistema UNIX, bem como dos compressores implementados.

4. BITM — Este conjunto considera arquivos de imagens no formato *bitmap* do sistema *XWindows*[6]. São vários tipos de imagens, incluindo fotografias e desenhos.

Os dados foram subdivididos em três categorias de comprimento, que são: PEQUENO, que considera arquivos de comprimento de até 10 KB, MÉDIO, que considera arquivos que variam de 10 a 100 KB, e GRANDE, com arquivos acima de 100 KB. A depender do tipo do dado, conseguiu-se um maior ou menor número

[5] *GNU-Emacs* é um editor de textos de domínio público distribuído pela *Free Software Foundation*, Inc.

[6] *X Window Systems* é uma marca registrada do M.I.T, Cambridge, Massachusetts.

Entropia						
Comprimento		Tipo				
		DOC	PROG	EXEC	BITM	Média
PEQUENO	1	4,83	4,55	4,51	3,39	4,32
	2	3,78	3,37	3,16	2,52	3,21
MÉDIO	1	4,90	4,72	4,84	2,92	4,35
	2	4,13	3,80	3,68	2,18	3,45
GRANDE	1	4,86	4,95	5,84	2,61	4,57
	2	4,21	4,18	4,66	2,08	3,78
Média	1	4,86	4,74	5,06	2,97	4,41
	2	4,04	3,78	3,83	2,26	3,48

Tabela 4.2: Entropia, medida em *bits/byte*, considerando-se todos os comprimentos e tipos de dados.

de arquivos em cada categoria. Ao todo, foram 207 arquivos, somando cerca de 10 MB. A tabela 4.1 apresenta a quantidade e o comprimento total dos arquivos.

Efetuamos o cálculo da *entropia* para os arquivos do conjunto de dados, a partir da fórmula (2.2) (*bits/byte*), enunciada na seção 2.2. Este cálculo considerou dois alfabetos de símbolos, um formado por caracteres do alfabeto ASCII e outro por duplas de caracteres. A tabela 4.2 apresenta os valores encontrados. Estes valores expressam uma média da entropia individual de cada arquivo do conjunto de dados. Ainda foi possível a determinação do tamanho do alfabeto dos símbolos ocorrendo nos arquivos. A tabela 4.3 expressa a média destes tamanhos. Os símbolos 1 e 2, nas tabelas, identificam uso de caracteres e duplas, respectivamente. O estabelecimento destas medidas possibilitou a seguinte avaliação das características dos dados.

Como era esperado, em todos os tipos de dados, a entropia para duplas é menor que para caracteres.

De um modo geral, excetuando-se os dados BITM, o valor da entropia cresce com o aumento do comprimento dos arquivos. Acredita-se que este aspecto é devido ao fato de que o maior comprimento dos arquivos, além de proporcionar um maior tamanho de alfabeto, possibilita uma distribuição mais uniforme dos símbolos. Para os dados BITM, ocorre exatamente o inverso. Uma explicação para este fato talvez seja o pequeno tamanho de alfabeto, mesmo em arquivos GRANDES, apresentado por este tipo de dado.

Observa-se ainda que os dados BITM são aqueles cuja entropia apresenta menores valores, sendo portanto os dados que devem possibilitar uma maior redução

Alfabeto de Símbolos						
Comprimento		Tipo				
		DOC	PROG	EXEC	BITM	Média
PEQUENO	1	75	69	169	40	88
	2	451	354	495	184	371
MÉDIO	1	88	86	248	39	115
	2	973	856	4.435	216	1.620
GRANDE	1	96	95	256	39	122
	2	1.804	1.577	11.885	244	3.878
Média	1	86	83	224	39	108
	2	1.076	926	5.605	215	1.956

Tabela 4.3: Tamanho médio do alfabeto de símbolos ocorrendo no conjunto de dados.

no comprimento dos arquivos. Os dados PROG têm entropia menor que os dados DOC. Os dados EXEC têm a maior entropia para caracteres, e no uso de duplas, exibem um valor menor que a entropia dos dados DOC.

4.2.2 Métodos

Os métodos implementados estão enumerados abaixo:

1. SF — Método de Shannon-Fano.
2. HUFEST — Método de Huffman estático.
3. HUFDIN — Método de Huffman dinâmico, com implementação sugerida por Knuth [Knu85].
4. ARIT — Método Aritmético, com implementação sugerida por Witten *et alii* [WNC87].
5. E-BSTW — Método proposto independentemente por Elias e Bentley *et alii* [Eli87] [BSTW86].
6. LZSS — Método Lempel-Ziv-1977 [ZL77], com modificações de Storer e Szymanski [SS82] e implementação sugerida por Bell [Bel86].
7. LZS — Método Lempel-Ziv-1978 [ZL78], com modificações sugeridas por Storer [Sto88].

Os compressores genéricos avaliados foram os seguintes: *pack*, *compact*, *compress* (4.0), PKPAK (3.61), PKZIP (1.10), ICE (1.14), LHA (2.13) e ARJ (2.10).

4.2.3 Medidas de Compressão

A compressão alcançada foi medida através da *taxa de compressão*, expressa pela fórmula (1.1) (%), enunciada na seção 1.1. Já que estamos apresentando uma comparação empírica, acreditamos que o leitor interessado em resultados práticos esteja muito mais acostumado com a taxa de compressão do que com outras medidas, até porque esta costuma ser a adotada pela maioria dos compressores genéricos.

Para ser possível uma comparação do desempenho dos métodos com o valor da *entropia*, também estaremos expressando esta medida em taxa de compressão. A entropia, usualmente expressa em *bits/byte*, e que fornece uma interpretação inversa a da taxa de compressão, passa a ser apresentada com base na fórmula: $(1 - (entropia/8))(\%)$. Considera-se assim, que o método atinge a entropia se a diferença entre a compressão alcançada e a entropia não excede 12%.

Alguns dos métodos permitiram uma distinção entre taxa de compressão *assintótica* e taxa *efetiva*. A compressão assintótica apenas considera o comprimento do texto comprimido. A compressão efetiva considera conjuntamente com este comprimento o espaço ocupado por informações necessárias à descompressão, tais como a árvore de prefixo para métodos estatísticos, a codificação de novos símbolos, etc. Obviamente, os valores de compressão apresentados para os compressores genéricos são efetivos.

Para a apresentação dos resultados, efetuou-se a média dos valores obtidos da compressão individual sobre cada arquivo do conjunto de dados.

4.3 Experimentos Individuais

A realização dos experimentos inicialmente consistiu em escolher uma faixa de parâmetros para os vários métodos. Para esta escolha, houve a necessidade da realização de experimentos preliminares, que foram efetuados com alguns arquivos do conjunto de dados. A seguir, iremos apresentar, para cada método, o conjunto de parâmetros considerados, ao mesmo tempo em que efetuamos uma avaliação sobre o seu desempenho individual.

As tabelas a serem expostas apresentam os resultados globais da compressão obtida para cada método. Os símbolos 1, 2 e P, em algumas tabelas, identificam que o método foi utilizado comprimindo caracteres, duplas de caracteres e cadeias de caracteres (palavras).

4.3.1 Método de Shannon-Fano (SF)

Este método foi implementado utilizando uma modelagem estática, com base no cálculo das freqüências dos símbolos ocorrendo no texto fonte. Duas versões foram consideradas. Uma que adota um alfabeto ASCII estendido de 256 caracteres e outra que utiliza duplas de caracteres. Efetuou-se duas medidas de compressão, a compressão assintótica, que considera o comprimento do texto comprimido sem a árvore de prefixo, e a compressão efetiva, que considera o espaço ocupado pela árvore de prefixo.

A tabela 4.4 apresenta os resultados de compressão para este método.

Observou-se que em todos os tipos e comprimentos de dados a compressão assintótica converge para a compressão determinada pela entropia, sendo que obviamente nunca a ultrapassa. A compressão efetiva, por sua vez, é sempre menor que a assintótica. Esta diferença acentua-se mais ainda quando os arquivos têm comprimento PEQUENO, ou quando duplas de caracteres são utilizadas.

De uma maneira geral, à medida que o comprimento do arquivo aumenta, a compressão efetiva converge para a assintótica. Este fato é observado mais fortemente para dados BITM, e não se observa para dados EXEC no uso de duplas. De qualquer forma, no uso de duplas, a diferença entre os dois valores de compressão ainda é significativa.

O fraco desempenho na compressão efetiva de dados PEQUENOS ou de duplas, e a diferença significativa entre os valores de compressão assintóticos e efetivos, pode ser explicado. Se o comprimento dos textos é muito pequeno, a representação da árvore de prefixo pode ser muito dispendiosa, principalmente para o alfabeto de duplas. Além disso, no uso de duplas de caracteres, à exceção dos dados BITM, os textos não foram suficientemente grandes para a obtenção de um ganho efetivo significativo, e conseqüentemente para a identificação de redundâncias.

Acredita-se que o fraco desempenho para a compressão de duplas dos dados EXEC é por este tipo de dado freqüentemente apresentar um tamanho grande de alfabeto. Ou seja, todos os símbolos do alfabeto previsto tendem a ocorrer no texto, e neste caso as observações acima se aplicam. Por outro lado, o bom desempenho para dados BITM, para quaisquer comprimentos de arquivos, deve-se a um tamanho de alfabeto muito pequeno, e constante.

4.3.2 Método de Huffman Estático (HUFEST)

A modelagem estática foi feita a partir do cálculo das freqüências dos símbolos ocorrendo no texto fonte. Considerou-se uma versão com um alfabeto de caracte-

SF							
Comprimento			Tipo				
			DOC	PROG	EXEC	BITM	Média
PEQUENO	1	(1 - entropia)	40	43	44	58	46
		tx. assint.	39	42	42	57	45
		tx. efetiva	35	38	35	56	41
	2	(1 - entropia)	53	58	61	69	60
		tx. assint.	52	57	60	68	59
		tx. efetiva	25	30	28	64	37
MÉDIO	1	(1 - entropia)	39	41	40	64	46
		tx. assint.	38	40	37	63	45
		tx. efetiva	38	40	37	62	44
	2	(1 - entropia)	48	53	54	73	57
		tx. assint.	48	52	53	72	56
		tx. efetiva	41	44	37	71	48
GRANDE	1	(1 - entropia)	39	38	27	67	43
		tx. assint.	39	37	26	66	42
		tx. efetiva	39	37	26	66	42
	2	(1 - entropia)	47	48	42	74	53
		tx. assint.	47	47	41	74	52
		tx. efetiva	46	45	29	73	48
Média	1	(1 - entropia)	39	41	37	63	45
		tx. assint.	39	40	35	62	44
		tx. efetiva	37	38	33	62	42
	2	(1 - entropia)	50	53	52	72	57
		tx. assint.	49	52	51	71	56
		tx. efetiva	37	40	31	69	44

Tabela 4.4: Resultados dos experimentos para o método SF, medidos pela taxa de compressão (%), considerando-se todos os comprimentos e tipos de dados.

res e outra versão com duplas de caracteres. Foi medida a compressão assintótica, sem considerar o espaço ocupado pelas árvores de prefixo, e a compressão efetiva.

A tabela 4.5 apresenta os resultados de compressão para o método. As observações de desempenho para este método são as mesmas relacionadas no método SF.

4.3.3 Método de Huffman Dinâmico (HUFDIN)

Na implementação do algoritmo dinâmico FGK [Knu85], considerou-se uma versão com um alfabeto de caracteres, uma versão com duplas de caracteres e uma versão com palavras.

Mediu-se a compressão efetiva, que considera o espaço ocupado pela codificação de novos símbolos (caracteres, duplas ou palavras) no texto, e a codificação assintótica, que não considera este espaço.

A tabela 4.6 apresenta os resultados de compressão de caracteres e duplas. De um modo geral, o desempenho obtido pelo método neste tipo de compressão é o mesmo relacionado nos métodos SF e HUFEST.

A versão com palavras foi implementada com o conceito de janela proposto por Knuth. Esta visou efetuar uma comparação com o método E-BSTW. As palavras consideradas foram alfanuméricas, formadas por letras do alfabeto romano e números, e não-alfanuméricas. Permitiu-se a variação de dois parâmetros: o *tamanho do dicionário de palavras* e o maior *comprimento da palavra*. O tamanho escolhido para a janela foi o mesmo do dicionário de palavras. Observações dos experimentos preliminares evidenciaram que o uso de uma janela de comprimento fixo — 2 KB é um bom valor — para dados BITM e EXEC, obtém praticamente o mesmo desempenho que o da janela adotada, e, para dados DOC e PROG, obtém desempenho um pouco inferior.

Escolheu-se as seguintes faixas de parâmetros: dicionário de palavras de 2^8 a 2^{15} e palavras de comprimentos máximos de 4 a 64 letras, os valores variando em potência de dois.

Para a determinação dos melhores parâmetros, realizamos o seguinte: inicialmente observamos os valores de compressão fixando o tamanho do dicionário e o comprimento da palavra, posteriormente observamos os parâmetros obtidos da melhor compressão individual de cada arquivo. Apresentamos a seguir as conclusões sobre os resultados obtidos.

HUFEST							
Comprimento			Tipo				
			DOC	PROG	EXEC	BITM	Média
PEQUENO	1	(1 - entropia)	40	43	44	58	46
		tx. assint.	39	43	43	57	46
		tx. efetiva	35	38	36	56	41
	2	(1 - entropia)	53	58	61	69	60
		tx. assint.	53	58	60	68	60
		tx. efetiva	25	30	28	64	37
MÉDIO	1	(1 - entropia)	39	41	40	64	46
		tx. assint.	38	41	38	63	45
		tx. efetiva	38	40	38	63	45
	2	(1 - entropia)	48	53	54	73	57
		tx. assint.	48	52	54	72	57
		tx. efetiva	41	44	37	72	49
GRANDE	1	(1 - entropia)	39	38	27	67	43
		tx. assint.	39	38	27	67	42
		tx. efetiva	39	38	26	67	42
	2	(1 - entropia)	47	48	42	74	53
		tx. assint.	47	48	42	74	53
		tx. efetiva	46	45	30	73	48
Média	1	(1 - entropia)	39	41	37	63	45
		tx. assint.	39	40	36	62	44
		tx. efetiva	37	39	33	62	43
	2	(1 - entropia)	50	53	52	72	57
		tx. assint.	49	53	52	71	56
		tx. efetiva	37	40	32	70	45

Tabela 4.5: Resultados dos experimentos para o método HUFEST, medidos pela taxa de compressão (%), considerando-se todos os comprimentos e tipos de dados.

HUFDIN							
Comprimento			Tipo				
			DOC	PROG	EXEC	BITM	Média
PEQUENO	1	(1 - entropia)	40	43	44	58	46
		tx. assint.	39	42	42	57	45
		tx. efetiva	35	38	37	56	42
	2	(1 - entropia)	53	58	61	69	60
		tx. assint.	52	57	60	68	59
		tx. efetiva	27	32	31	64	39
	P	tx. assint.	77	82	83	79	80
		tx. efetiva	43	53	40	74	53
MÉDIO	1	(1 - entropia)	39	41	40	64	46
		tx. assint.	38	41	38	63	45
		tx. efetiva	38	40	38	63	45
	2	(1 - entropia)	48	53	54	73	57
		tx. assint.	48	52	53	72	56
		tx. efetiva	42	45	38	72	49
	P	tx. assint.	71	79	81	83	79
		tx. efetiva	54	63	46	82	61
GRANDE	1	(1 - entropia)	39	38	27	67	43
		tx. assint.	39	38	26	67	42
		tx. efetiva	39	38	26	67	42
	2	(1 - entropia)	47	48	42	74	53
		tx. assint.	47	47	41	74	52
		tx. efetiva	46	45	31	73	49
	P	tx. assint.	68	75	75	84	75
		tx. efetiva	61	63	34	84	60
Média	1	(1 - entropia)	39	41	37	63	45
		tx. assint.	39	40	36	62	44
		tx. efetiva	37	39	34	62	43
	2	(1 - entropia)	50	53	52	72	57
		tx. assint.	49	52	51	71	56
		tx. efetiva	38	41	33	70	45
	P	tx. assint.	72	79	80	82	78
		tx. efetiva	53	60	40	80	58

Tabela 4.6: Resultados dos experimentos para o método HUFDIN, medidos pela taxa de compressão (%), considerando-se todos os comprimentos e tipos de dados.

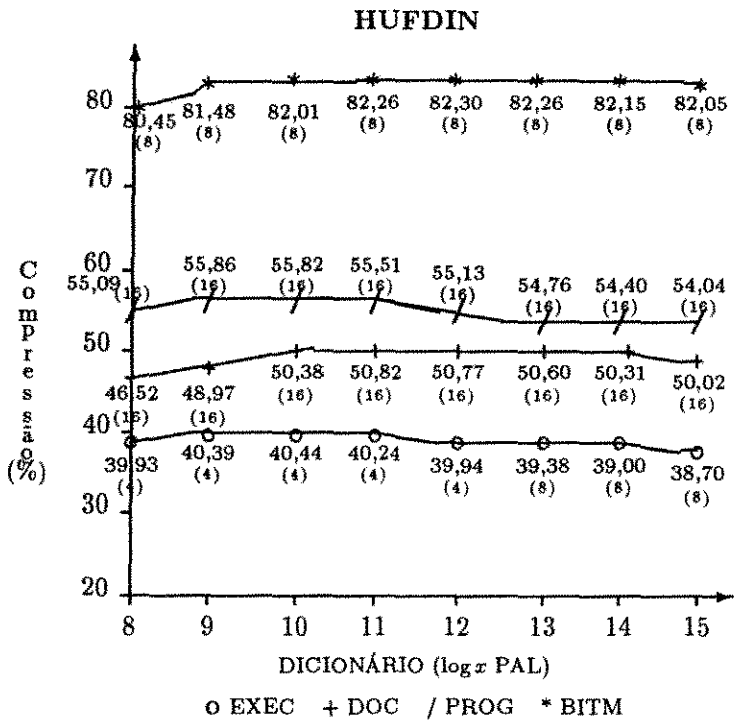


Figura 4.1: Compressão de HUFDIN para vários tamanhos da lista de palavras, considerando-se o conjunto total de dados.

O gráfico da figura 4.1 apresenta a variação de desempenho para os vários tamanhos de dicionário, considerando-se a compressão de todos os arquivos do conjunto de dados; os valores entre parênteses indicam o melhor valor encontrado para o comprimento da palavra. Nota-se que a perda entre escolher uma lista de tamanho menor — 2^{10} é um bom valor — é muito pouco significativa em relação á escolha de listas maiores.

Os melhores valores de comprimento de palavra foram de 16 e 32 para DOC e PROG e de 4 e 8 para EXEC e BITM. À exceção dos valores menores, que forneceram reduções significativas, o uso dos demais comprimentos não apresentou diferenças na compressão.

Os melhores valores para o tamanho da lista de palavras variaram com o comprimento do arquivo; quanto maior o comprimento, maior o valor. Estes valores situaram-se em 2^8 a 2^{10} para arquivos PEQUENOS, 2^{10} a 2^{12} para arquivos MÉDIOS e 2^{14} para arquivos GRANDES.

A média das melhores compressões de palavras de cada arquivo está expressa na tabela 4.6.

A separação do texto em palavras nem sempre é uma boa escolha para alguns tipos de fontes. A compressão assintótica apresentada não pretende estimar um limite de compressão a ser alcançado, mas antes estabelece um indicador da quantidade de palavras que ocorrem nos dados, ou se é interessante a realização de compressão com base nesta heurística. Observa-se pela tabela que os dados EXEC não se adequaram a esta heurística, e que, por sua vez, os dados BITM obtiveram um excelente desempenho, mesmo levando-se em consideração que o conjunto de palavras escolhidas não é o adequado a dados binários.

Na compressão efetiva a tabela evidencia o seguinte: à exceção dos dados BITM, a compressão de dados PEQUENOS e EXEC fica muito distante da entropia para duplas de caracteres. Para dados DOC e PROG, à medida que o comprimento do arquivo cresce a compressão vai sendo superior à entropia de duplas. O desempenho obtido com arquivos BITM é muito superior a entropia para duplas.

4.3.4 Método Aritmético (ARIT)

O método teve sua implementação baseada no algoritmo de Witten *et alii* [WNC87] e portanto precisou ter seu contador de freqüências de símbolos limitado. Escolheu-se uma grandeza de 15 *bits* para a maior freqüência dos símbolos no texto. Devido a esta limitação, só foi possível considerar um alfabeto de caracteres ASCII e não também de duplas, tais como nos demais métodos estatísticos. A modelagem utilizada foi a adaptativa.

Para este método, não há distinção entre taxa de compressão assintótica e efetiva. A tabela 4.7 apresenta os resultados obtidos.

As observações sobre o seu desempenho são basicamente as mesmas dos métodos SF, HUFEST e HUFDIN. Observou-se entretanto que para dados EXEC, nos maiores comprimentos de arquivos, o seu resultado foi superior à entropia. Acredita-se que este fato é devido à limitação do contador de freqüências. Portanto, o que a princípio parecia uma restrição possibilitou uma pequena vantagem deste método em relação aos demais estatísticos.

4.3.5 Método de Elias-Bentley (E-BSTW)

Implementou-se o método codificando-se um alfabeto de caracteres e de duplas de caracteres, da mesma forma que nos compressores estatísticos, para ser possível uma comparação com estes. Também foram codificadas palavras.

ARIT							
Comprimento			Tipo				
			DOC	PROG	EXEC	BITM	Média
PEQUENO	1	(1 - entropia)	40	43	44	58	46
		tx. efetiva	34	37	39	56	42
MÉDIO	1	(1 - entropia)	39	41	40	64	46
		tx. efetiva	38	40	40	63	45
GRANDE	1	(1 - entropia)	39	38	27	67	43
		tx. efetiva	39	38	30	67	44
Média	1	(1 - entropia)	39	41	37	63	45
		tx. efetiva	37	38	37	62	44

Tabela 4.7: Resultados dos experimentos para o método ARIT, medidos pela taxa de compressão (%), considerando-se todos os comprimentos e tipos de dados.

Na codificação de duplas e caracteres, a lista de símbolos já foi inicializada com o alfabeto inicial, portanto novos símbolos não precisaram ser codificados durante o processo. Conseqüentemente, não há distinção entre taxa assintótica e efetiva. Utilizou-se o código de Huffman dinâmico para a representação da posição do símbolo na lista.

A tabela 4.8 evidencia os seguintes resultados para o uso de caracteres e duplas. De um modo geral, à exceção dos dados BITM, os resultados obtidos ficaram muito abaixo da entropia. Este aspecto é observado mais fortemente na compressão de caracteres e duplas para dados PEQUENOS, e de duplas para dados MÉDIOS. O desempenho obtido para os dados BITM foi muito superior às duas entropias.

A versão com palavras considerou palavras alfanuméricas, formadas por letras do alfabeto romano e números, e não-alfanuméricas. Permitiu-se a variação de três parâmetros: o *tamanho da lista de palavras*, o *maior comprimento da palavra* e a *codificação de prefixo* utilizada para codificar caracteres das novas palavras, seu comprimento, e a sua posição na lista. Para a codificação de prefixo foram utilizadas três árvores de Huffman dinâmicas visando representar os três elementos citados. Como alternativa, códigos de Elias do tipo δ também foram utilizados para representar os dois últimos elementos.

Escolheu-se as seguintes faixas de parâmetros: lista de palavras com tamanhos de 2^7 a 2^{15} e palavras de comprimentos de 4 a 64 letras, os valores variando em potência de dois.

Para a determinação dos melhores parâmetros, realizamos o seguinte: inici-

E-BSTW							
Comprimento			Tipo				
			DOC	PROG	EXEC	BITM	Média
PEQUENO	1	(1 - entropia)	40	43	44	58	46
		tx. efetiva	29	33	33	64	40
	2	(1 - entropia)	53	58	61	69	60
		tx. efetiva	11	19	23	62	29
	P	tx. assint.	78	83	89	79	82
		tx. efetiva	41	54	50	74	55
		ELIAS	25	41	29	63	39
MÉDIO	1	(1 - entropia)	39	41	40	64	46
		tx. efetiva	34	37	36	69	44
	2	(1 - entropia)	48	53	54	73	57
		tx. efetiva	35	38	35	77	46
	P	tx. assint.	75	82	84	84	81
		tx. efetiva	51	63	46	83	61
		ELIAS	41	55	34	71	50
GRANDE	1	(1 - entropia)	39	38	27	67	43
		tx. efetiva	36	35	25	67	41
	2	(1 - entropia)	47	48	42	74	53
		tx. efetiva	44	42	30	79	49
	P	tx. assint.	70	80	80	85	79
		tx. efetiva	59	62	33	84	60
		ELIAS	52	57	18	73	50
Média	1	(1 - entropia)	39	41	37	63	45
		tx. efetiva	33	35	31	67	42
	2	(1 - entropia)	50	53	52	72	57
		tx. efetiva	30	33	29	73	41
	P	tx. assint.	74	82	84	83	81
		tx. efetiva	51	60	43	81	58
		ELIAS	39	51	27	69	47

Tabela 4.8: Resultados dos experimentos para o método E-BSTW, medidos pela taxa de compressão (%), considerando-se todos os comprimentos e tipos de dados.

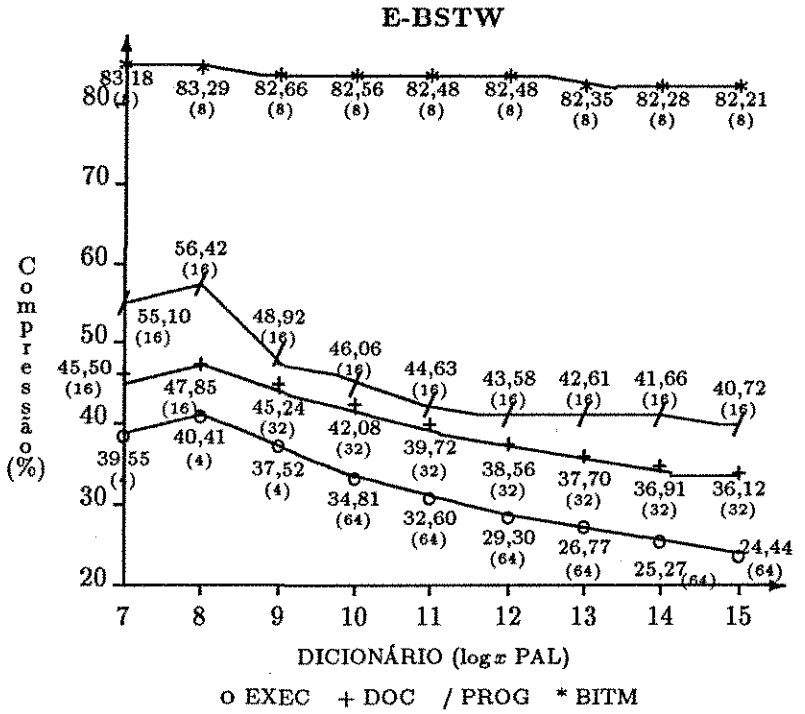


Figura 4.2: Compressão de E-BSTW para vários tamanhos da lista de palavras, considerando-se o conjunto total de dados.

almente observamos os valores de compressão fixando o tamanho do dicionário e o comprimento da palavra, posteriormente observamos os parâmetros obtidos da melhor compressão individual de cada arquivo. Apresentamos a seguir as conclusões sobre os resultados obtidos.

O gráfico da figura 4.2 apresenta a variação de desempenho para os vários tamanhos de dicionário, considerando-se a compressão de todos os arquivos do conjunto de dados; os valores entre parênteses indicam o melhor valor encontrado para o comprimento da palavra. Nota-se que não é interessante a utilização de listas de tamanho grande; isto porque com a codificação de maiores posições, a compressão adquirida passa a não ser significativa.

Verificou-se que os melhores valores para o comprimento das palavras foram de 16 e 32 para os dados PROG e DOC, 4 para os dados EXEC e de 4 e 8 para BITM. Para os demais comprimentos, à medida que os valores iam aumentando ocorria uma redução não significativa de cerca de 1% na taxa de compressão em

relação ao comprimento anterior.

Observou-se que o melhor valor para o tamanho da lista ficou em torno de 256, sendo que este valor tendia a aumentar com o aumento do comprimento dos arquivos. Estes valores situaram-se em 128 e 256 para arquivos PEQUENOS, 256 e 512 para arquivos MÉDIOS e 512 e 1024 para arquivos GRANDES.

No uso de palavras, a média das melhores compressões está expressa na tabela 4.8. Além da compressão efetiva, que considera o espaço ocupado pela codificação das novas palavras, e da compressão assintótica, que não considera este espaço, é apresentada a compressão efetiva utilizando os códigos de Elias (ELIAS), em substituição aos códigos de Huffman.

De um modo geral, as observações feitas para este método com o uso de palavras são as mesmas feitas para o método HUFDIN atuando em palavras.

Embora a utilização dos códigos de Elias permita uma implementação do método muito mais rápida e que utiliza menos memória, os valores de compressão obtidos foram cerca de 10% a 20% mais baixos comparativamente aos códigos de Huffman.

4.3.6 Método Lempel-Ziv-1977-Storer-Szymanski (LZSS)

O método LZSS [SS82] utiliza a implementação sugerida por Bell [Bel86]. O método permite uma variação do *tamanho da janela* sobre o texto fonte e do *maior padrão* para casamento. Na saída do seu código ainda foi possível avaliar a utilização da codificação dinâmica de Huffman. Neste caso, foram utilizadas três árvores de Huffman, uma para codificar os caracteres, outra para codificar a posição na janela e outra para codificar o comprimento do padrão.

Escolheu-se as seguintes faixas de parâmetros: tamanho da janela de 2^{10} a 2^{16} caracteres e comprimento do maior padrão de 8 a 256 caracteres, os valores variando em potência de dois.

Para a determinação dos melhores parâmetros, realizamos o seguinte: inicialmente observamos os valores de compressão fixando a janela e o comprimento do fator, posteriormente observamos os parâmetros obtidos da melhor compressão individual de cada arquivo. Apresentamos a seguir as conclusões sobre os resultados obtidos.

Observações dos experimentos preliminares evidenciaram que o uso de janelas maiores que o comprimento dos arquivos reduz ainda mais a compressão. Neste caso, este tipo de compressão não foi permitida nos experimentos. Uma vez que os comprimentos das janelas eram potências de 2, o seu maior valor não pôde exceder duas vezes o comprimento do arquivo. Devido à esta restrição, não é possível apresentar um resultado genérico — tal como nos dois métodos

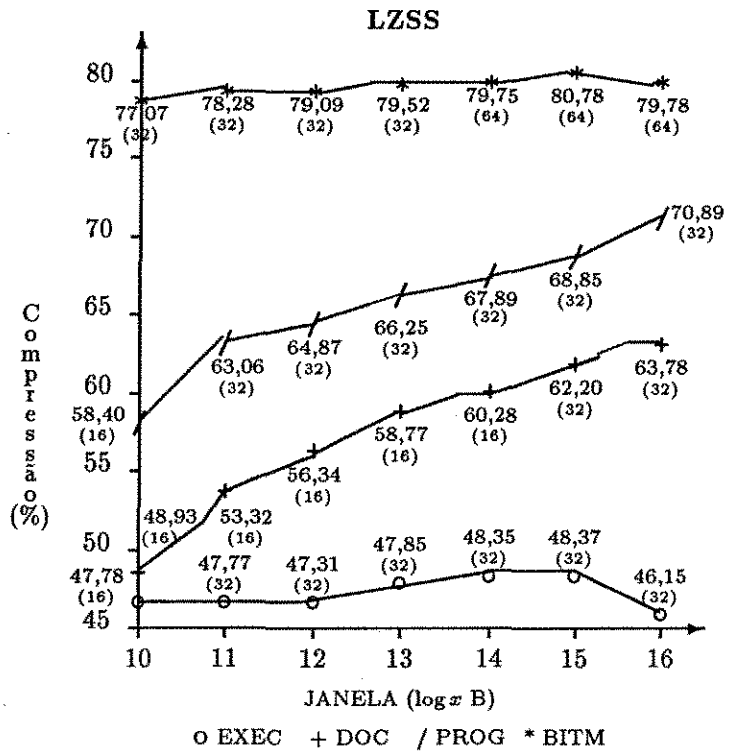


Figura 4.3: Compressão de LZSS para vários comprimentos de janela, considerando-se o conjunto total de dados.

anteriores — que avalie a compressão sobre todos os arquivos do conjunto de dados, variando-se o comprimento da janela. Desta maneira, apresentamos na figura 4.3 um gráfico que expressa para cada comprimento de janela a melhor compressão e comprimento de padrão, obtidos com apenas alguns arquivos do conjunto de dados. Assim, por exemplo, para a janela de 2^{16} caracteres, não está sendo considerada a compressão dos arquivos PEQUENOS.

Observou-se que os melhores comprimentos para os padrões situa-se em torno de 32, 16 e 64 para todos os tipos de dados. Bons resultados para os dados BITM também foram alcançados com 128 e 256 caracteres.

Para todos os dados, verificou-se que o tamanho da janela deve possuir valores proporcionais ao tamanho do arquivo. De um modo geral, para fontes PEQUENAS, os melhores valores de janela foram de 2^{10} e 2^{12}, para MÉDIAS, 2^{12} a 2^{15}, e GRANDES, 2^{16}.

LZSS							
Comprimento			Tipo				
			DOC	PROG	EXEC	BITM	Média
PEQUENO	2	(1 - entropia)	53	58	61	69	60
	P	tx. efetiva	48	59	49	69	56
		HUFDIN	45	58	48	63	54
MÉDIO	2	(1 - entropia)	48	53	54	73	57
	P	tx. efetiva	62	69	54	81	67
		HUFDIN	58	66	56	80	65
GRANDE	2	(1 - entropia)	47	48	42	74	53
	P	tx. efetiva	67	68	43	83	65
		HUFDIN	62	64	46	82	63
Média	2	(1 - entropia)	50	53	52	72	57
	P	tx. efetiva	59	66	49	78	63
		HUFDIN	55	63	50	75	61

Tabela 4.9: Resultados dos experimentos para o método LZSS, medidos pela taxa de compressão (%), considerando-se todos os comprimentos e tipos de dados.

Foi medida a compressão efetiva, que considera o espaço ocupado pela codificação dos caracteres no texto e dos *bits* indicadores, e a compressão efetiva utilizando a codificação de Huffman (HUFDIN). Não achamos correto enunciar uma codificação assintótica. A tabela 4.9 apresenta os resultados.

A tabela evidencia que para dados EXEC a compressão alcançada pelo método aproxima-se da entropia de duplas, à medida que o comprimento do arquivo cresce. Para os demais dados, seu desempenho é superior à entropia de duplas e cresce com o comprimento dos arquivos.

A codificação de Huffman na saída da compressão só consegue obter melhorias pouco significativas, de cerca de 4%, para arquivos EXEC muito GRANDES e através de janelas pequenas, quando seu desempenho é preferível à utilização de janelas maiores e sem a codificação de prefixo. De um modo geral, a tabela evidencia que o uso desta codificação para dados EXEC foi melhor (cerca de 2%) em relação à codificação fixa, considerando-se arquivos MÉDIOS e GRANDES. Para os demais dados a codificação obteve taxas inferiores.

4.3.7 Método de Lempel-Ziv-1978-Storer (LZS)

Método baseado na heurística de dicionário dinâmico proposta por Storer [Sto88]. Os padrões escolhidos para serem acrescentados ao dicionário basearam-se na heurística TP (todos os prefixos). Os códigos gerados foram de comprimento variável, i.é, acompanharam o crescimento do dicionário. A maior profundidade da *trie* que representa o dicionário foi de 256, ou seja, do tamanho do alfabeto ASCII inicial. Os experimentos preliminares evidenciaram que a limitação na altura da *trie* não foi uma grande restrição, inclusive para dados EXEC e BITM, um valor menor alcançava melhor compressão.

O método permite a variação do *tamanho do dicionário*, do *maior fator* contido neste e ainda da heurística de remoção das palavras no dicionário. Utilizou-se as heurísticas LRU (*Least Recently Used*), CLEAR e FREEZE. Na saída do seu código, ainda foi possível avaliar a utilização da codificação dinâmica de Huffman.

Escolheu-se as seguintes faixas de parâmetros: tamanho do dicionário de 2^{10} a 2^{16} caracteres, comprimento do maior padrão de 1 a 256 caracteres — os valores variando em potência de dois, além das heurísticas de remoção do dicionário.

Verificou-se que o uso das heurísticas de remoção são equiparadas, sendo que as melhores em ordem são LRU, CLEAR e FREEZE, diferenciando-se em cerca de 5% uma em relação a outra. Para dados BITM seu desempenho é praticamente o mesmo. Os valores de compressão a serem apresentados são relativos à heurística LRU.

Para a determinação dos melhores parâmetros, realizamos o seguinte: inicialmente observamos os valores de compressão fixando o tamanho do dicionário e o comprimento do fator, posteriormente observamos os parâmetros obtidos da melhor compressão individual de cada arquivo. Apresentamos a seguir as conclusões sobre os resultados obtidos.

O gráfico da figura 4.4 apresenta a variação de desempenho para os vários tamanhos de dicionário, considerando-se a compressão dos arquivos do conjunto de dados; os valores entre parênteses indicam o melhor valor encontrado para o comprimento do fator. Devido à limitação no tamanho do dicionário, decidiu-se efetuar a mesma restrição na compressão adotada pelo método LZSS. Ou seja, não foram utilizados tamanhos de dicionário maiores que o comprimento dos textos. Desta forma, os valores apresentados para os tamanhos de dicionário no gráfico expressam somente a compressão de alguns arquivos do conjunto de dados.

Observamos que variados comprimentos de padrão foram utilizados. Para fontes DOC e PROG valores médios, 16 a 64 foram os mais utilizados. Para fontes EXEC, valores menores, 4 a 8 foram mais freqüentes. Os dados BITM apresentaram valores bastante variados, desde 1 a 256.

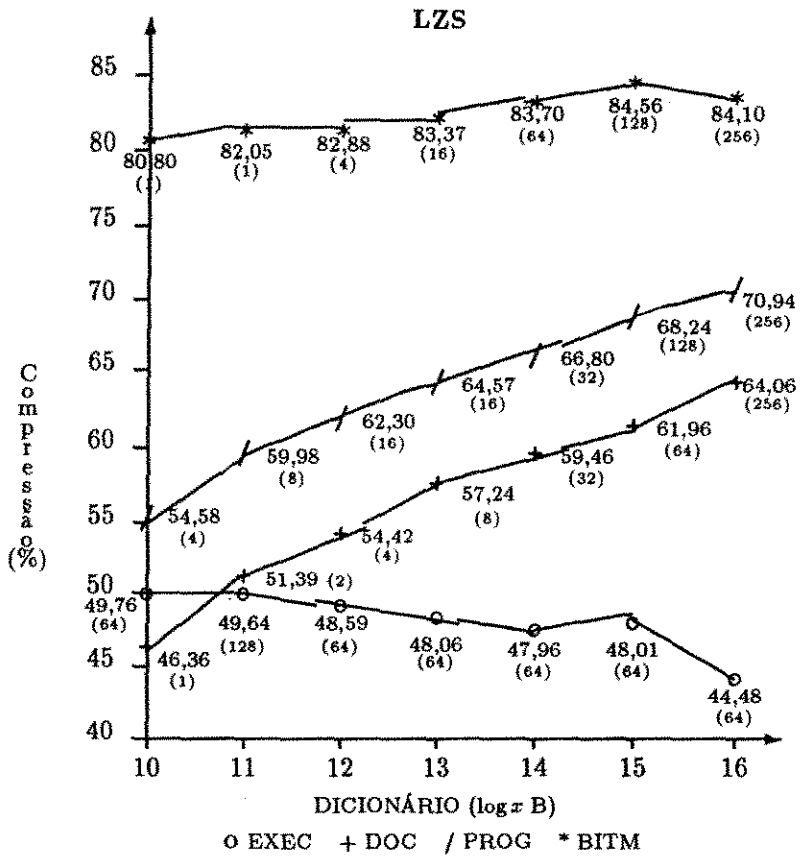


Figura 4.4: Compressão de LZS para vários comprimentos do dicionário, considerando-se o conjunto total de dados.

LZS							
Comprimento			Tipo				
			DOC	PROG	EXEC	BITM	Média
PEQUENO	2	(1 - entropia)	53	58	61	69	60
	P	tx. efetiva	48	57	51	73	57
		HUFDIN	35	45	41	68	47
MÉDIO	2	(1 - entropia)	48	53	54	73	57
	P	tx. efetiva	61	68	55	84	67
		HUFDIN	50	59	55	81	61
GRANDE	2	(1 - entropia)	47	48	42	74	53
	P	tx. efetiva	66	68	43	86	66
		HUFDIN	58	59	45	83	62
Média	2	(1 - entropia)	50	53	52	72	57
	P	tx. efetiva	58	64	50	81	63
		HUFDIN	48	54	47	78	57

Tabela 4.10: Resultados dos experimentos para o método LZS, medidos pela taxa de compressão (%), considerando-se todos os comprimentos e tipos de dados.

Para fontes do tipo DOC e PROG o tamanho do dicionário apresentou valores proporcionais ao comprimento do arquivo. Entretanto, para fontes do tipo EXEC e BITM pode-se considerar um tamanho fixo de dicionário que depende do comprimento do arquivo. Os melhores valores foram os seguintes: para dados PEQUENOS: 2^{10} e 2^{11}, para MÉDIOS: 2^{12} a 2^{14}, para GRANDES: 2^{14} e 2^{15}.

As observações sobre o desempenho deste método na compressão das fontes são as mesmas que para o método LZSS. Para este método, a compressão efetiva não apresenta gastos (*overhead*). Os resultados obtidos estão expressos na tabela 4.10.

Excetuando-se uma pequena melhora na compressão de arquivos GRANDES, com comprimentos superiores a 1 MB, a codificação de Huffman na saída da compressão não obteve nenhuma vantagem.

4.3.8 Compressores Genéricos

Em todos os compressores utilizamos parâmetros necessários à máxima compressão. Desta forma, para o *compress* adotamos um comprimento máximo de código de 16 *bits*; para o ARJ utilizamos a opção "-jm", de realização máxima de compressão; para o PKZIP adotou-se a opção "-ex", também para a máxima de

compressão.

Em relação às modalidades de compressão adotadas pelos compressores genéricos, observamos o seguinte: o utilitário PKPAK utilizou somente as modalidades que utilizam o LZW, que são a *Crunching* e a *Squashing*; o utilitário PKZIP utilizou apenas a modalidade *Imploding*, que adota o LZSS; em arquivos muitos pequenos, o *pack* não efetuou compressão alguma.

4.4 Comparação entre os Métodos

Nesta seção, apresentamos uma comparação entre os desempenhos obtidos pelos métodos; ao mesmo tempo, efetuamos uma correlação entre os resultados alcançados e o que foi previsto na apresentação dos mesmos, realizada nos Capítulos anteriores.

As tabelas 4.11 a 4.18 apresentam os melhores resultados de compressão efetiva alcançados tanto pelos métodos como pelos compressores genéricos. As tabelas 4.11 a 4.16 apresentam os resultados pelos comprimentos dos dados. As tabelas 4.17 e 4.18 apresentam os resultados globais. Os valores em negrito expressam a melhor compressão dentre as demais; quando a diferença entre as melhores compressões é insignificativa — abaixo de 1% — são destacadas em negrito todas elas.

As tabelas demonstram que, de um modo geral, há uma equivalência nos resultados dos métodos estatísticos, comprimindo caracteres e duplas.

Embora tenhamos visto na seção 3.1.1 que o método SF não constrói códigos ótimos, observa-se que o seu resultado é praticamente o mesmo do método HUFEST. A diferença entre os dois não excede 1%, e com o aumento do comprimento dos arquivos tende a diminuir.

O resultado do método HUFDIN nunca esteve abaixo do desempenho dos métodos estáticos (SF e HUFEST); inclusive, a compressão de duplas para dados PEQUENOS foi relativamente superior ao desempenho daqueles métodos. A explicação para este fato talvez seja a não necessidade de representação da árvore de prefixo. A compressão de duplas dos EXEC, para qualquer comprimento, também foi superior. Possivelmente, em parte isto se deve ao grande tamanho de alfabeto exibido por este tipo de fonte.

O método ARIT foi o que, no geral, obteve a melhor compressão dentre os métodos estatísticos; inclusive, para dados EXEC, a diferença foi significativa. Este fato comprova que a limitação no contador de freqüências não resultou em perda de compressão.

À exceção dos dados BITM, o resultado alcançado pelo método E-BSTW na compressão de caracteres e duplas ficou bem abaixo do resultado dos demais estatísticos. De qualquer forma, conforme previsto na seção 3.2.1, esta diferença não excedeu duas vezes o resultado do método de Huffman. O desempenho do E-BSTW para os dados BITM foi muito superior ao dos estatísticos.

Comparando os resultados dos métodos HUFDIN e E-BSTW atuando em palavras observamos que eles são equivalentes. Entretanto, em alguns casos, há diferenças; o E-BSTW se destaca na compressão de dados BITM, enquanto o HUFDIN comprime melhor dados DOC. Para os dados BITM, a heurística de utilização de palavras obtém excelentes resultados. A equivalência de desempenho entre os dois métodos nos leva a considerar que se fosse aplicado o método HUFDIN com janelas na compressão de caracteres em dados BITM, ele iria obter resultado semelhante ao E-BSTW.

Os resultados demonstraram, conforme previa Bell (seção 3.2.3), que na prática, há uma equivalência de desempenho entre o LZSS e o LZS. Portanto, apesar da heurística sofisticada, o LZS não conseguiu sobrepujar o LZSS. Em alguns casos, entretanto, aconteceram diferenças significativas. O LZS, para compressão de dados EXEC e principalmente BITM, foi superior; o LZSS, por sua vez, foi superior em dados DOC e principalmente PROG.

As tabelas evidenciam a superioridade dos compressores Lempel-Ziv sobre todos os demais métodos. Observa-se, entretanto, que para dados BITM o desempenho destes métodos e dos compressores atuando em palavras é praticamente o mesmo.

Em relação aos compressores genéricos, tal qual observamos nos métodos, os compressores estatísticos foram os de pior desempenho.

Os compressores que utilizam o método LZW, a saber *compress* e PKPAK, têm desempenho semelhante. Excetuando-se para os dados EXEC, onde a compressão é superior, este desempenho também é equivalente ao dos métodos HUFDIN e E-BSTW com palavras. O desempenho com os dados EXEC, entretanto, ficou abaixo da entropia de duplas.

O compressor ICE, que praticamente utiliza o mesmo método do LZSS, obteve um resultado muito superior nos dados EXEC. Acreditamos que isto se deve ao uso da compressão de Huffman. Este compressor tem desempenho semelhante ao PKZIP, que na modalidade mais utilizada, i.é, *imploding*, utiliza o LZSS. O desempenho do ICE é melhor para dados PEQUENOS e do PKZIP é melhor para maiores comprimentos.

Os compressores LHA e ARJ obtiverem as melhores compressões dentre todos os métodos e compressores genéricos. O ARJ foi relativamente superior, prin-

cipalmente em dados DOC. Entretanto, no geral, a diferença foi insignificativa. Uma conclusão importante sobre este resultado é que, ao que parece, o método LZFG, no qual o LHA se fundamenta, foi o método que proporcionou a melhor compressão.

Pelos resultados obtidos da compressão dos métodos, concluímos que as fontes BITM apresentam uma forte localidade de referência. De tal maneira, que mesmo métodos menos sofisticados, como o LZW, HUFDIN e E-BSTW com palavras conseguem obter resultados de compressão semelhantes aos melhores valores. Os dados EXEC são os que testaram mais fortemente os compressores, sendo necessárias heurísticas, do tipo codificação de Huffman na saída do código Lempel-Ziv, para um ganho de compressão.

Como conclusão das observações aqui realizadas, apresentamos o diagrama abaixo, que estabelece uma relação de precedência entre os métodos. Esta relação não é válida para a compressão de dados BITM. Os símbolos desta precedência estabelecem o seguinte: $\ll$ indica que o desempenho do método está muito abaixo do outro; $<$ indica que o resultado do método está sempre abaixo; $\leq$ indica que o método, em algumas situações, obtém desempenho igual ou superior.

Estat. $\ll$ E-BSTW, HUFDIN (P) $\leq$ LZW $\ll$ LZSS, LZS $\ll$ LZSS (estat.) $<$ LZFG (HUFEST)

Entre os compressores uma relação também se estabelece.

Pack, compact $\ll$ *Compress,* PKPAK $\ll$ PKZIP, ICE $<$ LHA, ARJ

Para as fontes de dados, pode-se também efetuar uma relação de precedência de acordo com a sua capacidade de compressão. Os símbolos da precedência têm o mesmo significado anterior, somente que indicam que a fonte possibilita um maior ou menor compressão.

EXEC $\ll$ DOC $<$ PROG $\ll$ BITM

PEQUENOS						
Métodos		Dados				
		DOC	PROG	EXEC	BITM	Média
(1 - Entropia)	1	40	43	44	58	46
	2	53	58	61	69	60
SF	1	35	38	35	56	41
	2	25	30	28	64	37
HUFEST	1	35	38	36	56	41
	2	25	30	28	64	37
HUFDIN	1	35	38	37	56	42
	2	27	32	31	64	39
	P	43	53	40	**74**	53
ARIT	1	34	37	39	56	42
E-BSTW	1	29	33	33	64	40
	2	11	19	23	62	29
	P	41	54	42	**74**	53
LZSS		**48**	**59**	49	69	**56**
LZS		**48**	57	**51**	**73**	**57**

Tabela 4.11: Resultados dos experimentos dos métodos, medidos pela taxa de compressão (%), considerando-se os dados PEQUENOS.

PEQUENOS					
Compressores Genéricos	Dados				
	DOC	PROG	EXEC	BITM	Média
pack	30	36	37	56	40
compact	36	39	39	56	43
compress	43	51	45	71	52
PKPAK	46	53	46	71	54
PKZIP	54	64	54	71	61
ICE	55	66	55	72	62
LHA	**58**	**68**	**57**	**75**	**64**
ARJ	**58**	**68**	**57**	**75**	**64**

Tabela 4.12: Resultados dos experimentos dos compressores genéricos, medidos pela taxa de compressão (%), considerando-se os dados PEQUENOS.

MÉDIOS						
Métodos		Dados				
		DOC	PROG	EXEC	BITM	Média
(1 - Entropia)	1	39	41	40	64	46
	2	48	53	54	73	57
SF	1	38	40	37	62	44
	2	41	44	37	71	48
HUFEST	1	38	40	38	63	45
	2	41	44	37	72	49
HUFDIN	1	38	40	38	63	45
	2	42	45	38	72	49
	P	54	63	46	82	61
ARIT	1	38	40	40	63	45
E-BSTW	1	34	37	36	69	44
	2	35	38	35	77	46
	P	51	63	46	**83**	61
LZSS		**62**	**69**	**54**	81	**67**
LZS		**61**	**68**	**55**	**84**	**67**

Tabela 4.13: Resultados dos experimentos dos métodos, medidos pela taxa de compressão (%), considerando-se os dados MÉDIOS.

MÉDIOS					
Compressores Genéricos	Dados				
	DOC	PROG	EXEC	BITM	Média
pack	38	40	38	63	45
compact	38	40	38	63	45
compress	55	61	49	82	62
PKPAK	56	62	51	83	63
PKZIP	67	**74**	60	83	71
ICE	64	72	59	83	69
LHA	67	**75**	**62**	**85**	**72**
ARJ	**69**	**75**	**62**	**85**	**73**

Tabela 4.14: Resultados dos experimentos dos compressores genéricos, medidos pela taxa de compressão (%), considerando-se os dados MÉDIOS.

GRANDES						
Métodos		Dados				
		DOC	PROG	EXEC	BITM	Média
(1 - Entropia)	1	39	38	27	67	43
	2	47	48	42	74	53
SF	1	39	37	26	66	42
	2	46	45	29	73	48
HUFEST	1	39	38	26	67	42
	2	46	45	30	73	48
HUFDIN	1	39	38	26	67	42
	2	46	45	31	73	49
	P	61	63	34	84	60
ARIT	1	39	38	30	67	44
E-BSTW	1	36	35	25	67	41
	2	44	42	30	79	49
	P	59	62	33	84	60
LZSS		**67**	**68**	**43**	83	**65**
LZS		**66**	**68**	**43**	**86**	**66**

Tabela 4.15: Resultados dos experimentos dos métodos, medidos pela taxa de compressão (%), considerando-se os dados GRANDES.

GRANDES					
Compressores Genéricos	Dados				
	DOC	PROG	EXEC	BITM	Média
pack	39	38	26	67	42
compact	39	38	26	67	42
compress	62	62	37	**85**	62
PKPAK	57	59	40	**85**	60
PKZIP	68	71	49	84	68
ICE	65	69	49	84	66
LHA	68	71	**52**	**86**	69
ARJ	**71**	**73**	**53**	**86**	**71**

Tabela 4.16: Resultados dos experimentos dos compressores genéricos, medidos pela taxa de compressão (%), considerando-se os dados GRANDES.

GLOBAL						
Métodos		Dados				
		DOC	PROG	EXEC	BITM	Média
Entropia	1	39	42	37	65	46
	2	50	56	52	73	57
SF	1	37	39	33	64	43
	2	37	35	33	72	44
HUFEST	1	37	39	34	64	43
	2	37	36	33	72	44
HUFDIN	1	37	39	34	64	44
	2	38	37	35	72	45
	P	51	57	41	82	58
ARIT	1	37	38	37	64	44
E-BSTW	1	33	35	32	68	42
	2	29	27	32	77	41
	P	49	57	41	**83**	58
LZSS		**58**	**63**	**50**	81	**63**
LZS		**58**	61	**51**	**84**	**63**

Tabela 4.17: Resultados dos experimentos dos métodos, medidos pela taxa de compressão (%), considerando-se o conjunto total de dados.

GLOBAL					
Compressores Genéricos	Dados				
	DOC	PROG	EXEC	BITM	Média
Pack	36	35	34	64	42
Compact	37	40	35	64	44
Compress	52	54	45	83	59
PKPAK	53	56	47	83	60
PKZIP	64	68	56	82	67
ICE	62	68	55	82	67
LHA	**65**	**70**	**58**	**85**	**69**
ARJ	**66**	**71**	**58**	**85**	**70**

Tabela 4.18: Resultados dos experimentos dos compressores genéricos, medidos pela taxa de compressão (%), considerando-se o conjunto total de dados.

Bibliografia

[Abr63] N. Abramson. *Information Theory and Coding.* McGraw-Hill, New York, 1963.

[Abr89] D. M. Abrahamson. An adaptive dependency source model for data compression. *Communications of the ACM*, 32(1):77–83, janeiro de 1989.

[AG84] A. Apostolico e R. Giancarlo. Pattern matching machine implementation of a fast test for unique decipherability. *Information Processing Letters*, 18(3):155–158, março de 1984.

[Bel86] T. C. Bell. Better OPM/L text compression. *IEEE Transactions on Communications*, 34(12):1176–1182, dezembro de 1986.

[Bel87] T. C. Bell. *A unifying theory and improvements for existing approaches to text compression.* PhD thesis, Dept. of Computer Science, University of Canterbury, New Zealand, 1987.

[BM85] J. L. Bentley e C. McGeoch. Amortized analyses of self-organizing sequential search heuristics. *Communications of the ACM*, 28(4):404–411, abril de 1985.

[BSTW86] J. L. Bentley, D. D. Sleator, R. E. Tarjan, e V. K. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29(4):320–330, abril de 1986.

[BWC89] T. C. Bell, I. H. Witten, e J. G. Cleary. Modeling for text compression. *ACM Computing Surveys*, 21(4):557–591, dezembro de 1989.

[CH84] G. V. Cormack e R. N. S. Horspool. Algorithms for adaptive Huffman codes. *Information Processing Letters*, 18(3):159–165, março de 1984.

[CH87] G. V. Cormack e R. N. S. Horspool. Data compression using dynamic Markov modelling. *Computer Journal*, 30(6):541–550, dezembro de 1987.

[Cor85] G. V. Cormack. Data compression on a database system. *Communications of the ACM*, 28(12):1336–1342, dezembro de 1985.

[Cro89] M. Crochemore. Data compression with substitution. In M. Gross e D. Perrin, editores, *Eletronic Dictionaries and Automata in Computational Linguistics*, pp. 1–16. Springer-Verlag, Berlin, 1989. Lecture Notes in Computer Science, 377.

[CW84] J. G. Cleary e I. H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 32(4):396–402, abril de 1984.

[Eli75] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, IT-21(2):194–203, março de 1975.

[Eli87] P. Elias. Interval and recency rank source coding: two on-line adaptive variable-length schemes. *IEEE Transactions on Information Theory*, IT-33(1):3–10, janeiro de 1987.

[Eve79] S. Even. *Graph Algorithms*. Computer Science Press, Potomac, Maryland, 1979.

[Fal73] N. Faller. An adaptive system for data compression. In *Proc. 7th Asilomar Conference on Circuits, Systems and Computers*, pp. 593–597, Pacific Grove, California, 1973.

[Fan49] R. M. Fano. *Transmission of Information*. M.I.T. Press, Cambridge, Mass., 1949.

[FG89] E. R. Fiala e D. H. Greene. Data compression with finite windows. *Communications of the ACM*, 32(4):490–505, abril de 1989.

[FMP83] A. S. Fraenkel, M. Mor, e Y. Perl. Is text compression by prefixes and suffixes practical? *Acta Informatica*, 20(4):371–389, dezembro de 1983.

[Gal78] R. G. Gallager. Variations on a theme by Huffman. *IEEE Transactions on Information Theory*, IT-24(6):668–674, novembro de 1978.

[Ham80] R. W. Hamming. *Coding and Information Theory.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1980.

[Han89] G. Hansel. Estimation of the entropy by the Lempel-Ziv method. In M. Gross e D. Perrin, editores, *Eletronic Dictionaries and Automata in Computational Linguistics*, pp. 51–65. Springer-Verlag, Berlin, 1989. Lecture Notes in Computer Science, 377.

[HC83] R. N. Horspool e G. V. Cormack. Data compression based on token recognition. Manuscrito não publicado, outubro de 1983.

[HC87] R. N. Horspool e G. V. Cormack. A locally adaptive data compression scheme. *Communications of the ACM*, 30(9):792–794, setembro de 1987.

[Hel87] G. Held. *Data Compression: Techniques and Applications, Hardware and Software Considerations.* John Wiley & Sons, New York, segunda edição, 1987.

[HL90] D. S. Hirschberg e D. A. Lelewer. Efficient decoding of prefix codes. *Communications of the ACM*, 33(4):449–459, abril de 1990.

[Huf52] D. A. Huffman. A method for the construction of minimun-redundancy codes. *Proceedings of the I.R.E.*, 40(9):1098–1101, setembro de 1952.

[Jun90] Robert K. Jung. *ARJ Software & Manual (2.10 v. – 1991).* Norwood, MA, 1990. Informações técnicas retiradas do arquivo "technote.txt".

[Kat86] P. W. Katz. *PKware File Compression Programs — PKPAK 3.61 v. (1988).* PKWARE Inc., Glendale, WI, 1986. Informações técnicas retiradas do arquivo "appnote.txt".

[Kat89] P. W. Katz. *PKware File Compression Programs — PKZIP 1.1 v. (1990).* PKWARE Inc., Glendale, WI, 1989. Informações técnicas retiradas do arquivo "appnote.txt".

[KBD89] S. T. Klein, A. Bookstein, e S. Deerwester. Storing text retrieval systems on CD-ROM: compression and encryption considerations. *Transactions on Information Systems*, 7(3):230–245, julho de 1989.

[Knu73] D. E. Knuth. *The Art of Computer Programming*, volume 3 de *Sorting and Searching.* Addison-Wesley, Reading, MA, 1973.

[Knu85] D. E. Knuth. Dynamic Huffman coding. *Journal of Algorithms*, 6(2):163–180, junho de 1985.

[Lan83] G. G. Langdon. A note on the Ziv-Lempel model for compressing individual sequences. *IEEE Transactions on Information Theory*, IT-29(2):284–287, março de 1983.

[Lan84] G.G. Langdon. An introduction to arithmetic coding. *IBM Journal of Research and Development*, 28(2):135–149, março de 1984.

[LH87] D. A. Lelewer e D. S. Hirschberg. Data compression. *ACM Computing Surveys*, 19(3):261–296, setembro de 1987.

[Lle87] J. A. Llewellyn. Data compression for a source with Markov characteristics. *Computer Journal*, 30(2):149–156, abril de 1987.

[LZ76] A. Lempel e J. Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, IT-22(1):75–81, janeiro de 1976.

[Mak89] E. Mäkinen. On implementing two adaptive data-compression schemes. *Computer Journal*, 32(3):238–240, março de 1989.

[McC76] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, abril de 1976.

[Mor68] D. R. Morrison. PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.

[MP85] D. R. McIntyre e M. A. Pechura. Data compression using static Huffman code-decode tables. *Communications of the ACM*, 28(6):612–616, junho de 1985.

[MW85] V. S. Miller e M. N. Wegman. Variations on a theme by Ziv and Lempel. In A. Apostolico e Z. Galil, editores, *Combinatorial Algorithms on Words*, pp. 131–140. Springer-Verlag, Berlim, 1985. NATO Advanced Science Institutes, Series F, Vol. 12.

[Pas76] R. Pasco. *Source coding algorithms for fast data compression.* PhD thesis, Stanford University, Stanford, CA, 1976.

[Ris76] J. Rissanen. Generalized Kraft inequality and arithmetic coding. *IBM Journal of Research and Development*, 20:198–203, maio de 1976.

[Riv76] R. Rivest. On self-organizing search heuristics. *Communications of the ACM*, 19(2):63–67, fevereiro de 1976.

[RL79] J. Rissanen e G.G. Langdon. Arithmetic coding. *IBM Journal of Research and Development*, 23(2):149–162, março de 1979.

[RL81] J. Rissanen e G.G. Langdon. Universal modeling and coding. *IEEE Transactions on Information Theory*, 27(1):12–23, janeiro de 1981.

[RPE81] M. Rodeh, V. R. Pratt, e S. Even. Linear algorithm for data compression via string matching. *Journal of the ACM*, 28(1):16–24, janeiro de 1981.

[Rub79] F. Rubin. Arithmetic stream coding using fixed precision registers. *IEEE Transactions on Information Theory*, 25(6):672–675, novembro de 1979.

[Rya80] B. Y. Ryabko. Data compression by means of a "book stack". *Prob. Inf. Transm.*, 16(4), 1980.

[Rya87] B. Y. Ryabko. A locally adaptive data compression scheme. *Communications of the ACM*, 30(9):792, setembro de 1987.

[Sch64] E. S. Schwartz. An optimum encoding with minimum longest code and total number of digits. *Information & Control*, 7(1):37–44, março de 1964.

[Sha48] C. E. Shannon. A mathematical theory of communication. *Bell Syst. Tech. J.*, 27:398–403, julho de 1948.

[Sie88] A. Siemiński. Fast decoding of the Huffman codes. *Information Processing Letters*, 26(5):237–241, janeiro de 1988.

[SS82] J. A. Storer e T. G. Szymanski. Data compression via textual substitution. *Journal of the ACM*, 29(4):928–951, outubro de 1982.

[ST85] D. D. Sleator e R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):625–686, julho de 1985.

[Sto88] J. A. Storer. *Data Compression - Methods and Theory.* Computer Science Press, Rockville, MD, 1988.

[SW49] C. E. Shannon e W. Weaver. *The Mathematical Theory of Communication.* University of Illinois Press, Urbana, Ill, 1949.

[TMD+84] S. Thomas, J. McKie, S. Davies, K. Turkowski, J. Woods, e J. Orost. *Compress, Programa e Documentação (v. 4.0 - 1985)*, 1984.

[Vit87] J. S. Vitter. Design and analysis of dynamic Huffman codes. *Journal of the ACM*, 34(4):825–845, outubro de 1987.

[Vit89] J. S. Vitter. Dynamic Huffman coding. *ACM Transactions on Mathematical Software*, 15(2):158–167, junho de 1989.

[WC88] I. H. Witten e J. G. Cleary. On the privacy afforded by adaptive text compression. *Computers and Security*, 4(7):397–408, agosto de 1988.

[Wel84] T. A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, junho de 1984.

[WNC87] I. H. Witten, R. M. Neal, e J. G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, junho de 1987.

[Yos88] Haruyasu Yoshizaki. *ICE User's Manual and Software (1.14 v. - 1989)*, 1988. Informações técnicas retiradas do conjunto de fontes.

[Yos91] Haruyasu Yoshizaki. *LHA User's Manual and Software (2.13 v. - 1991)*, 1991. Informações técnicas retiradas do próprio manual.

[Zip90] M. Zipstein. *Les Méthodes de Compression de textes: Algorithmes et Performances.* PhD thesis, Université Paris VII, Laboratoire Informatique Théorique et Programmation, Institut Blaise Pascal, março de 1990.

[ZL77] J. Ziv e A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, maio de 1977.

[ZL78] J. Ziv e A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, setembro de 1978.