**Compressão Adaptativa
usando
Codificação Aritmética**

Marcelo Ris

DISSERTAÇÃO APRESENTADA
AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO
PARA OBTENÇÃO DO GRAU DE MESTRE
EM
MATEMÁTICA APLICADA

Área de Concentração: Ciência da Computação
Orientador: Prof. Dr. Imre Simon

Durante a elaboração deste trabalho, o autor recebeu apoio financeiro do CNPq

-São Paulo, maio de 1996-

Compressão Adaptativa usando Codificação Aritmética

Este exemplar corresponde à redação
final da dissertação devidamente corrigida
e defendida por Marcelo Ris e aprovada
pela comissão julgadora.

São Paulo, 20 de maio de 1996.

Banca examinadora:

- Prof. Dr. Imre Simon (orientador) - IME-USP
- Prof. Dr. José Augusto Ramos Soares - IME-USP
- Prof. Dr. Jaime Szwarcfiter - UFRJ

Agradecimentos

Aos meus pais, a quem devo toda a minha educação,
Aos meus irmãos, Márcia e Fernando, a quem estimo muito,
À Neninha, a quem devo todo meu crescimento,
Ao Prof. Imre Simon, de quem tanto aprendi nesses anos em que pude contar com a sua valiosa orientação,
À Banca Examinadora, pela cuidadosa leitura e pelas inúmeras correções,
À Fabiana e à Verena, pela revisão do texto,
À Ana Amélia, pela revisão do texto em inglês,
Ao Fábio, por me ajudar tanto a operar as estações,
Ao Ricardo, por me ajudar no texto,
A todos os meus colegas de estudo que me ajudaram muito nestes anos de estudo,
À Vera, por ter me apoiado ao longo deste trabalho,
A todos meus amigos que me incentivaram,
A todos os professores do instituto,
E ao CNPq, pela bolsa,

o meu *sincero agradecimento*.

Marcelo Ris.

Resumo

A técnica de compressão denominada por compressão aritmética é analisada do ponto de vista prático ao longo deste trabalho. A primeira parte analisa dois métodos de codificação aritmética, um com o uso de multiplicações e divisões, e um segundo sem o uso destas operações. Na segunda parte são apresentados os métodos para a estimativa de probabilidades assim como um novo método, o qual aparentemente consegue melhores resultados do que os já existentes. Uma longa lista de análises comparativas entre os vários métodos de compressão usando um codificador aritmético é apresentada, além da comparação entre o novo método e os compressores utilizados atualmente. Para todas estas análises, foi desenvolvido um programa, o qual está disponível para futuras pesquisas.

Abstract

The technique of compression named arithmetic compression is analyzed from a practical point of view throughout this work. The first part of this thesis analyzes two methods of arithmetic coding, one using divisions and multiplications, and another one, which makes no use of these operations. The methods to estimate probabilities are presented in the second part of this work, as well as a new method, which apparently gets better results than those already known. A long list of comparative analyses among the various methods of compression using an arithmetic coder is given, besides a comparison between the new method and the other compressors used nowadays. A computer program was developed for every one of these analyses and it is available for future researches.

Conteúdo

1	Codificação Aritmética	3
1.1	História	4
1.2	Definições iniciais	6
1.3	O código de Huffman	8
1.4	O código aritmético	12
1.5	Implementando o Código Aritmético	16
1.5.1	Codificação e decodificação incrementais	18
1.5.2	O Problema da Decodificação	21
1.5.3	Tratamento de Overflow e Underflow	23
1.5.4	Eficiência do código aritmético	29
1.6	Uma segunda implementação	31
1.6.1	Implementação para um alfabeto binário	34
1.6.2	Implementando para um alfabeto não-binário	37
1.6.3	Eficiência do código	41
2	Modelos Adaptativos	47
2.1	Classes de equivalência	51
2.2	Modelos semi-adaptativos	54
2.2.1	Método exato	57
2.2.2	Método aproximado	58
2.2.3	Método fixo	60
2.2.4	Análise comparativa	61
2.3	Modelos adaptativos	64
2.3.1	Método adaptativo A	65
2.3.2	Método adaptativo B	66
2.3.3	Análise comparativa	66
2.4	Modelos adaptativos de contexto-finito	71

2.4.1	Mistura de estimativas	72
2.4.2	Probabilidade de escape A	75
2.4.3	Probabilidade de escape B	76
2.4.4	Probabilidade de escape C	77
2.4.5	Probabilidade de escape R	78
2.4.6	Exclusão	79
2.4.7	Divisão de frequências	82
2.4.8	Tamanho do alfabeto	85
2.4.9	Algoritmos adaptativos	87
2.4.10	Implementação dos modelos	87
2.4.11	Análise comparativa	92
2.5	O programa ppm	96

A	Conjunto de arquivos	100
----------	-----------------------------	------------

Introdução

Nosso trabalho foi motivado pela vontade de melhorar a taxa de compressão atingida hoje pelos compressores existentes, ou seja, simplesmente comprimir mais sem estar preocupado com eficiência no tempo gasto de processamento. Para isto, o trabalho é basicamente experimental e não leva a fundo características teóricas, as quais poderão ser analisadas em um trabalho posterior. Partindo deste ponto de vista, o trabalho cumpre a risca seu objetivo, onde além de analisar a fundo técnicas muito utilizadas obtém um método que possui melhores resultados práticos que os métodos existentes. Devemos salientar que os conceitos relativos a “entropia”, “quantidade de informação”, e outros usados no campo das compressões foram descritos de um ponto de vista intuitivo e não pretenderam atingir um nível de tratamento matemático mais aprofundado. Este nível de tratamento utilizado no trabalho é, porém, muito utilizado por autores que fazem apenas análises experimentais do assunto.

Dentro de todas as técnicas de compressão estudadas ao longo de nossa pesquisa nos interessamos por uma que nos pareceu ser a que obtinha melhores resultados no comprimento do código obtido. Este processo é denominado por compressão aritmética e se baseia em codificar probabilidades, ou seja, a cada instante estima probabilidades aos símbolos possíveis e a partir destas constrói o código. Para a análise deste processo dividimos nosso trabalho em dois capítulos: *codificação aritmética* e *modelos adaptativos*.

No primeiro capítulo analisamos apenas a codificação aritmética, e seu processo de codificar probabilidades, para isto analisamos dois algoritmos: um que utiliza no processo multiplicações e divisões, e um segundo que não requer estas operações, sendo assim mais utilizado para implementações em “hardware”.

No segundo capítulo tratamos de estudar a parte mais importante na compressão aritmética que é a atribuição de probabilidades. Para esta tarefa várias técnicas foram analisadas, sendo que a atenção foi mais voltada para os modelos adaptativos de contexto-finito. Dentro destes modelos, um novo método por nós desenvolvido parece obter melhores resultados do que os já existentes. Neste capítulo existe um grande número de tabelas comparativas entre os vários métodos, inclusive uma que analisa o desempenho do nosso compressor e os já existentes no mercado.

Para as análises ao longo do trabalho, foi desenvolvido um programa que

está disponível para pesquisa, o qual é totalmente parametrizado, podendo executar as inúmeras técnicas de compressão encontradas ao longo do trabalho. O programa e este trabalho podem ser conseguidos via ftp no endereço <ftp.ime.usp.br> no diretório pub/mris.

Devemos salientar que este trabalho foi adaptado em grande parte do trabalho de Bell, Cleary e Witten, em [19], para o qual algumas alterações foram apresentadas.

Capítulo 1

Codificação Aritmética

A compressão objetiva a partir de uma mensagem com um comprimento qualquer, transformar esta em uma outra de comprimento menor, sendo que esta segunda possua a mesma capacidade de compreensão. Para ilustrar este fenômeno, imaginemos que deseja-se comprimir um livro escrito em português de 100 páginas, e o mecanismo usado para tal seja retirar todos os artigos e pontuações do mesmo. Como resultado teremos um livro com um tamanho menor e com a “mesma” capacidade de entendimento. A transformação de uma mensagem em uma outra é denominada de *codificação*, e o termo *código* é empregado para designar o resultado. No exemplo, o livro sem artigos e pontuações é um código do livro original.

Neste trabalho estamos interessados em um tipo especial de compressão, a *compressão aritmética*. Este tipo de compressão se baseia em codificar probabilidades, isto é, transforma a mensagem em um código de comprimento inversamente proporcional a sua probabilidade. O grande problema deste tipo de compressão é estimar estas probabilidades, sendo assim, o trabalho foi dividido em duas partes. No primeiro capítulo daremos uma visão geral de como se comporta a codificação aritmética, analisando duas implementações para a mesma. No segundo capítulo, o qual está a base do nosso trabalho, analisaremos como se estima probabilidade para uma mensagem, e as várias técnicas existentes, inclusive uma nova por nós introduzida.

1.1 História

Abramson [1] dividiu todos os códigos em duas classes : código em *bloco*, e código em *não-bloco*. O termo “código em bloco” é caracterizado por cada símbolo ser representado por uma palavra-código e a concatenação destas palavras-código para cada símbolo de uma mensagem formam a mensagem codificada. Os códigos em bloco se subdividem em sub-classes tais como: códigos *não-singulares*, onde todas palavras-código são distintas, e códigos *instantâneos* para os quais a condição necessária e suficiente é a *propriedade dos prefixos*: “nenhuma palavra-código é prefixo de outra”. Como exemplo deste tipo de código, tomemos um texto que contenha somente as vogais $\{a, e, i, o, u\}$, um código instantâneo para este texto pode ser obtido pela substituição de cada vogal por uma sequência de 0's e 1's, por exemplo, $\{a = 00, e = 01, i = 10, o = 110, u = 111\}$. Este tipo de código é *univocamente decodificável*, pois, pela propriedade dos prefixos, partindo-se do começo do código somente um símbolo pode ser decodificado por vez.

Dentre todos os códigos em bloco, o código de maior interesse no campo da compressão é sem dúvida o *código compacto*, aquele cujo comprimento médio é menor ou igual a todos os códigos *univocamente decodificáveis*, existentes para um mesmo alfabeto e fonte. Aparentemente, o código compacto obtido pelo método de Huffman [7] é, entre todos os métodos de codificação sem ruído, o mais usual no que se refere à compressão. Este método, porém, muitas vezes requer *extensão do alfabeto* para conseguir uma boa compressão, ou seja, aumentar o número de símbolos do alfabeto transformando cada símbolo em uma palavra do mesmo alfabeto, por exemplo o uso de sílabas ao invés de apenas letras para caracterizar alfabetos. Devemos concluir então que os códigos em bloco, apesar de sua simplicidade e conhecida teoria de decodificação, não são satisfatórios para cobrir todos requisitos de uma “boa” compressão.

Tentando corrigir os defeitos dos códigos em bloco, surgem então, os códigos em não-bloco, os quais possuem a propriedade de não transformar cada símbolo em uma palavra-código específica de comprimento fixo, e sim em transformar cada símbolo em várias palavras-código de comprimentos variáveis ao longo do processo. Com isto, podemos assumir que os códigos em não-bloco transformam cada símbolo em uma palavra-código de comprimento igual à média de todas as palavras-código usadas para codificá-lo, ou seja, podemos dizer que a palavra-código possui comprimento “fracionário”

neste caso. No campo das classes de códigos em não-bloco, dois tipos de códigos apareceram nos anos 60 e começo de 70: o *código de Elias*, o qual mesmo não sendo sido publicado Abramson [1] faz uma rápida descrição, e os *códigos enumerativos* [5, 16]. Embora ambos precisassem de precisão aritmética ilimitada, não necessitavam de extensão do alfabeto para atingir compressão ótima.

Em 1976, Rissanen[16] e Pasco[14] descobrem novas classes de códigos em não-bloco, onde o cálculo poderia ser feito com precisão aritmética finita e em tempo linear. O código de Rissanen é LIFO (*last-in first-out*) e produz o código em ordem reversa. Já, o código de Pasco, diferentemente, é FIFO (*first-in first-out*) e mantém todo o código em memória, até que o último símbolo seja codificado, permitindo o deslocamento em toda palavra-código.

Rubin[18], Guazzo[6] e Rissanen e Langdon[17] aparecem na mesma época com os conceitos modernos de codificação aritmética: precisão aritmética fixa, operação incremental(símbolo a símbolo) e tipo FIFO. A mesma idéia foi descoberta independentemente por muitos pesquisadores.

Muitos estudos foram feitos no campo da codificação aritmética, sendo basicamente divididos em códigos para alfabetos binários e para alfabetos com mais de dois símbolos. Para o caso de alfabetos binários um código aritmético muito simples foi implementado por Langdon e Rissanen[12] em 1982.

Em 1981, Langdon[9] desenvolve um tutorial no campo da codificação aritmética, e, em 1984, é publicada uma versão mais atualizada[10], a qual inclui a história do método. Em 1987, Ian Witten, Radford M. Neal & John G. Cleary[8] obtém uma implementação completa do mesmo.

Recentemente, foi estudada por Rissanen e Mohiuddin[15] uma técnica de codificação aritmética que não usa multiplicações nem divisões. Esta técnica é baseada no código para alfabeto binário[12], porém, se aplica também a alfabetos com mais de dois símbolos. Uma implementação foi obtida por Radford M. Neal em 1991.

Ao longo deste capítulo, estudaremos duas implementações da codificação aritmética. A primeira [8], usada em nossa própria implementação, requer o uso de multiplicações e divisões. Já a segunda [15] é mais recente e não requer o uso destas operações, sendo assim, facilmente implementável em *hardware*.

1.2 Definições iniciais

Inicialmente, definiremos alguns conceitos que serão usadas ao longo deste trabalho.

Definição 1: Um *alfabeto* $A = \{a_1, a_2, \dots, a_m\}$ é um conjunto de $m \geq 1$ símbolos.

Definição 2: Uma *palavra* w sobre A é uma concatenação finita de símbolos de A . Por exemplo: $A = \{a, b, c, d\}$ e $w = cabad$.

Definição 3: Uma *fonte de informação* F é aquela que gera símbolos de A obedecendo a um modelo de probabilidade¹, ou seja, a cada instante i , a fonte gera um símbolo a_k com probabilidade $p(a_k, i)$, sendo que a soma das probabilidades de cada símbolo de A no instante i vale 1, ou seja, $\sum_{1 \leq k \leq m} p(a_k, i) = 1$. Denominaremos (a_k, i) por *instância* do instante i . Um exemplo de fonte é um gerador de palavras em português.

Definição 4: Uma função o é uma *ordenação instantânea sobre* A , se ela é uma bijeção de A no conjunto $\{1, 2, \dots, m\}$, a cada instante i . O *índice* de um símbolo de A é o valor de o aplicado ao mesmo. Vamos denotar por $y = o^{-1}$, a função inversa da ordenação o . No decorrer do trabalho ao nos referirmos a um símbolo de A , usaremos simplesmente o seu índice (a função o aplicada a ele) em A .

Definição 5: Uma *mensagem* $M = x_1x_2\dots x_t$ é uma palavra sobre A que pode ser gerada por F . Vale a pena notarmos que uma palavra w sobre A pode nunca ser gerada por F , logo nem sempre é uma mensagem sobre A . Se tomarmos o exemplo de F sendo um gerador de palavras em português, então a palavra “movie” não será uma mensagem gerada por F , já a palavra “cinema” é uma mensagem gerada por F .

Definição 6: A *quantidade de informação* I de uma instância é equivalente ao espaço ocupado pelo inverso de sua probabilidade em uma certa base. Por exemplo: na base 2, $I(a_k, i) = \lg(1/p(a_k, i)) = -\lg p(a_k, i)$ bits. A quantidade de informação de uma mensagem $M = x_1x_2\dots x_t$ é dada pela soma da quantidade de informação das instâncias que a compõem, isto é $I(M) = \sum_{i \leq t} I(x_i, i)$. A quantidade de informação de uma instância (ou mensagem) também pode ser entendida como o comprimento mínimo que um código sobre um alfabeto B pode ter. No nosso caso, como a quantidade de informação foi obtida sobre a base 2, então ela indica o menor comprimento

¹A definição de modelos será vista no próximo capítulo.

de um código sobre um alfabeto binário. Poderíamos expandir a idéia para códigos sobre alfabetos de tamanho $b > 2$, e para estes casos a quantidade de informação seria calculada usando logaritmos na base b . Ao longo deste trabalho usaremos o alfabeto binário $B = \{0, 1\}$ para representar os códigos, sendo assim, utilizaremos logaritmos na base 2 para o cálculo da quantidade de informação.

Inicialmente, iremos supor uma fonte F que gera símbolos seguindo um *modelo fixo* de probabilidades, isto é, a probabilidade de uma instância independe do instante i , logo $p(a_k, i) = p(a_k)$. A partir desta suposição, iremos definir um novo conceito, denominado por *entropia*, e que apesar do nome um tanto esquisito é o principal conceito no campo da compressão.

Definição 7: A *entropia* $S(F)$ de uma fonte fixa F sobre um alfabeto A é igual à soma da quantidade de informação de cada símbolo a_k de A multiplicada pela sua respectiva probabilidade $p(a_k)$, isto é:

$$S(F) = - \sum_{1 \leq k \leq m} p(a_k) \lg p(a_k).$$

Podemos entender a entropia como sendo a quantidade de informação “média” que uma fonte possui, ou seja, um limite inferior para o comprimento médio dos códigos relativos a cada instância gerada pela fonte. A unidade da entropia é expressa pela unidade do alfabeto do código dividida pela do alfabeto da mensagem. Se estivermos codificando caracteres ASCII sobre o alfabeto binário, a unidade da entropia será *bits/char*. Para uma fonte não fixa, onde as probabilidades dos símbolos variam conforme o instante do processo, o cálculo da entropia varia. Para este caso teríamos uma entropia para cada instante i do processo, que corresponde a uma distribuição de probabilidades. Para estes casos uma maneira de calcular $S(F)$ seria verificar as distribuições de probabilidades possíveis D_j , obter suas probabilidades $p(D_j)$ e calcular a sua entropia $S(D_j)$. Com isto teríamos que: $S(F) = \sum_j p(D_j) S(D_j)$. Como a entropia $S(F)$ é a quantidade de informação média que a fonte F possui, então ela nos fornece um valor mínimo para o comprimento de um código sobre um certo alfabeto B (no nosso caso binário) relativo a uma mensagem M gerada por F .

Definição 8: Um código C sobre um alfabeto B de uma mensagem M sobre um alfabeto A gerada por uma fonte F é *ótimo* se o comprimento médio relativos aos símbolos de M sobre C é igual a entropia de F . Isto equivale a $S(F) = |C|/|M|$, onde $|C|$ é o comprimento do código C sobre o alfabeto B , isto é, o número de símbolos de B contidos em C ; e $|M|$ é o comprimento de M sobre A . Um exemplo: seja F uma fonte com entropia 4

bits/char, então para uma mensagem de comprimento de 100 caracteres um código ótimo deve ter comprimento de 400 bits (ou 50 caracteres, supondo que cada caracter é representado por 8 bits).

1.3 O código de Huffman

Para o entendimento do código aritmético faz-se útil a revisão de como se dá a construção do *código de Huffman*[7], o qual possui a propriedade de codificar cada símbolo em uma cadeia de bits (no caso de usarmos o alfabeto binário para a codificação), podendo ser classificado, assim, como código em bloco. Veremos mais adiante que o código aritmético não transforma um símbolo em apenas uma cadeia de bits, mas guarda alguma informação adicional para o próximo símbolo. Sendo assim, podemos dizer que o código de Huffman transforma um símbolo em um código de comprimento inteiro e o código aritmético em um código de comprimento fracionário.

Para a codificação utilizando o alfabeto binário, Huffman utiliza uma árvore binária denominada de *árvore de Huffman*, a qual é construída a partir das probabilidades de ocorrência dos símbolos na mensagem a ser codificada. A árvore de Huffman T é uma árvore binária composta por nós, onde cada nó de T é formado por dois ponteiros: *no.esquerdo* para o filho esquerdo, e *no.direito* para o filho direito; uma certa probabilidade (*no.prob*) e um identificador (*no.cont*) que contém o símbolo que ele representa.

O processo de construção de T é feito, iterativamente, da seguinte maneira: seja P um conjunto de sub-árvores, no início P possui m sub-árvores com um nó cada, representando cada símbolo do alfabeto A com suas respectivas probabilidades. A cada passo, um novo nó r é criado, sendo ele a raiz de uma nova sub-árvore em P . O seguinte procedimento descreve o processo:

enquanto $componentes(P) > 1$
 $r.esquerdo = menor(P)$;
 $r.direito = menor(P)$;
 $r.prob = r.esquerdo.prob + r.direito.prob$;
 $P = P + \{r\}$;

onde $componentes(P)$ é uma função que retorna o número de sub-árvores em P , e $menor(P)$ retorna um ponteiro para a sub-árvore de P que possui em

sua raiz a menor probabilidade retirando, logo em seguida, esta sub-árvore do conjunto P . A árvore de Huffman T é a árvore resultante quando o processo é concluído. Como exemplo vamos supor o alfabeto $A = \{a, e, i, o, \bullet\}$, e um modelo fixo de probabilidades indicado na tabela 1.1. A figura 1.1 mostra, em cada passo, o conjunto P .

Símbolo	Probabilidade	Intervalo (P)	$-\lg \text{prob}$	Código de Huffman
a	0.4	$[0, 0.4)$	1.32	0
e	0.3	$[0.4, 0.7)$	1.73	10
i	0.1	$[0.7, 0.8)$	3.32	110
o	0.1	$[0.8, 0.9)$	3.32	1110
$\bullet$	0.1	$[0.9, 1.0)$	3.32	1111

Tabela 1.1: Exemplo de um modelo fixo para o alfabeto $\{a, e, i, o, \bullet\}$

Construída a árvore, cada símbolo é codificado pela cadeia binária que corresponde ao caminho em T , da raiz até o nó que o representa, sendo que quando percorrermos um filho à esquerda em T concatenamos 0, e quando percorrermos um filho à direita, concatenamos 1 ao código. O código de Huffman para uma mensagem M é obtido pela concatenação dos códigos obtidos para cada símbolo de M .

Podemos provar que, para uma fonte fixa onde cada instância possui probabilidade potência de 2, o código de Huffman obtém para cada instância (a_k, i) uma palavra-código sobre o alfabeto binário de comprimento igual a $I(a_k, i)$ bits. Para isto, basta notarmos que a construção da árvore T , para instâncias com estas probabilidades é feita de modo a manter cada folha, que representa uma instância (a_k, i) , a uma distância da raiz de comprimento igual a $-\lg p(a_k, i)$, que equivale à sua quantidade de informação. Deste modo, toda a mensagem gerada por esta fonte é codificada com um comprimento igual à sua quantidade de informação, e com isto para este tipo de fonte, o código de Huffman é ótimo.

Utilizamos o alfabeto binário para caracterizar o código obtido pela codificação de Huffman, porém, o código de Huffman pode gerar códigos em qualquer alfabeto de tamanho $m \geq 2$. Para isto, seria necessária a construção análoga de uma árvore onde cada nó pudesse possuir até m filhos. A otimalidade do código, neste caso, ocorre quando cada símbolo possuir probabilidade potência de m .

Figura 1.1: Construção da árvore de Huffman.

O código de Huffman sobre o alfabeto binário é ótimo para fontes fixas que geram símbolos com probabilidades potência de 2. O que vale a pena verificarmos agora é o que aconteceria com o código de Huffman sobre o alfabeto binário se a fonte F não gerasse símbolos com probabilidades potência de 2?

Analizando a codificação de Huffman para símbolos com probabilidades $p(a_k)$ diferentes de potência de 2, verificamos que as palavras-código obtidas possuem comprimento igual a quantidade de informação relativa a uma fonte F' que gera símbolos com probabilidade potência de 2, ou seja, se n_k são os comprimentos das palavras-código obtidas para os símbolos a_k , então estas palavras-código geram um código ótimo para a fonte que possui probabilidades $q(a_k)$ iguais a 2^{-n_k} . Podemos provar com isto, que estas mesmas palavras-código não geram um código ótimo para as atuais probabilidades $p(a_k)$, a proposição a seguir nos mostra isto.

Proposição 1 *Sejam $p_1, p_2, \dots, p_m$ as probabilidades dos símbolos $a_1, a_2, \dots, a_m$ gerados pela fonte F . Sejam $q_1, q_2, \dots, q_m$, onde cada $q_i = 2^{-n_i}$, e n_i é o comprimento da palavra-código obtida pela árvore de Huffman para o símbolo a_i .*

$\sum_{1 \leq i \leq m} p_i = 1$, pela definição de fonte.

$\sum_{1 \leq i \leq m} q_i = 1$, pode ser demonstrado por indução na construção da árvore T .

Logo, $-\sum_{1 \leq i \leq m} p_i \lg p_i \leq -\sum_{1 \leq i \leq m} p_i \lg q_i$, com igualdade sse $p_i = q_i$.

Prova: Sabemos que o logaritmo é uma função convergente, em outras palavras $\ln x$, sempre fica abaixo de sua tangente. Considerando a tangente em $x = 1$, obtemos $\ln x \leq x - 1$, com igualdade sse $x = 1$. Logo $\ln(q_i/p_i) \leq q_i/p_i - 1$ com igualdade sse $p_i = q_i$. Multiplicando esta inequação por p_i , e somando sobre todos os i 's temos que:

$\sum_{1 \leq i \leq m} p_i \ln \frac{q_i}{p_i} \leq \sum_{1 \leq i \leq m} (q_i - p_i)$, com igualdade se todo $q_i = p_i$.

Como o segundo termo vale 0, e multiplicando por $\lg e$, temos:

$\sum_{1 \leq i \leq m} p_i \lg q_i - \sum_{1 \leq i \leq m} p_i \lg p_i \leq 0$, com igualdade se todo $q_i = p_i$. CDQ.

A partir desta proposição podemos verificar que uma pequena variação nas probabilidades faz com que o código de Huffman não obtenha resultado ótimo.

1.4 O código aritmético

A codificação de Huffman obtém, para cada instância (a_k, i) , uma palavra-código de comprimento inteiro, o qual é uma aproximação da sua quantidade de informação ($\approx -\lg p(a_k, i)$). A codificação aritmética por sua vez, tentando corrigir esta aproximação feita pelo código de Huffman, surge com o objetivo de transformar cada instância em uma palavra-código de comprimento equivalente a sua quantidade de informação I , o que nem sempre é um número inteiro. A não-integridade do comprimento da palavra-código pode ser observada pela média ponderada dos comprimentos das palavras-código obtidas por um símbolo ao longo do processo. Veremos que um símbolo será codificado em cada instante por uma palavra-código de comprimento distinto, sendo que interpretaremos (no sentido de comparar com o código de Huffman) que este símbolo é codificado por uma palavra-código de comprimento igual à média ponderada das palavras-código obtidas ao longo do processo.

A partir de uma mensagem M do alfabeto A , o código é obtido recursivamente, com o objetivo de obter para cada instância (a_k, i) uma palavra-código de comprimento igual a sua quantidade de informação, ou seja, $-\lg p(a_k, i)$ bits. O logaritmo é na base 2, pois estamos trabalhando com um alfabeto binário para a codificação e com isso a unidade de codificação é o bit. Devemos levar em conta que como $p(a_k, i)$ nem sempre é potência de 2 existem instâncias que necessitarão de uma palavra-código de comprimento não-inteiro.

O código aritmético se baseia em projetar as probabilidades das instâncias no intervalo *real* unitário $[0, 1)$. Para isto, temos uma nova função $P(a_k, i) = \sum_{1 \leq j \leq k} p(a_j, i)$ chamada de probabilidade acumulativa. Esta função faz com que cada instância ocupe um sub-intervalo do intervalo real unitário $[0, 1)$. O método é executado iterativamente para cada instância, sendo que, a cada instante, o intervalo é restringido ao sub-intervalo correspondente à instância processada, e uma nova projeção de probabilidades é executada no novo intervalo. Finalizadas todas as instâncias, o código aritmético é um número real pertencente ao intervalo resultante.

Para melhor compreensão, vejamos um exemplo: seja o alfabeto $A = \{a, e, i, o, \bullet\}$, o modelo fixo da tabela 1.1 e a mensagem $M = eeio\bullet$. Inicialmente o codificador possui o intervalo real $[0, 1)$. Após receber o primeiro símbolo, e , o intervalo é restringido para o sub-intervalo $[0.4, 0.7)$ correspondente a este símbolo. O segundo símbolo e projeta seu intervalo $[0.4, 0.7)$ ao

intervalo resultante, obtendo assim $[0.52, 0.61)$. Como o tamanho do intervalo anterior era de 0.3 e o novo símbolo tem tamanho 0.3, então o novo intervalo possui tamanho igual à $0.3 \times 0.3 = 0.09$. Para o próximo símbolo i , o intervalo alocado é $[0.7, 0.8)$, que quando aplicado ao novo intervalo faz com que este diminua para $[0.583, 0.592)$. Sendo assim, a codificação da mensagem é construída do seguinte modo :

Inicialmente		$[0,$	$1)$
entrou símbolo :	e	$[0.4,$	$0.7)$
	e	$[0.52,$	$0.61)$
	i	$[0.583,$	$0.592)$
	o	$[0.5902,$	$0.5911)$
	•	$[0.59101,$	$0.5911)$

Na figura 1.2, uma representação gráfica tenta mostrar o mecanismo de codificação. No item (a), as divisões das barras verticais mostram o espaço reservado a cada símbolo dentro do intervalo corrente. Somente 3 iterações são mostradas nesta figura, pois o intervalo fica tão pequeno que fica imperceptível. No item (b), para cada intervalo resultante temos uma expansão na dimensão da figura.

Ao finalizar a codificação, o código obtido é apenas um número real r pertencente ao intervalo final resultante, no exemplo este número poderia ser 0.59101.

Vejam agora, como funciona o mecanismo de decodificação. Vamos supor que o decodificador receba o número real $r = 0.59101$. No início ele possui o intervalo $[0, 1)$ e os sub-intervalos relativos a cada símbolo como os da tabela 1.1, logo, o decodificador pode deduzir imediatamente que o primeiro símbolo é o símbolo e , pois a ele corresponde o sub-intervalo $[0.4, 0.7)$ onde r se localiza. O próximo passo reduz o intervalo a $[0.4, 0.7)$, projeta as probabilidades neste intervalo, e procura localizar r em algum novo sub-intervalo. Este processo é executado iterativamente, porém, um problema surge ao decodificador: *quando o processo deve ser finalizado?*. Uma maneira de resolver este problema é manter um símbolo especial como “fim” de mensagem, e no momento em que o decodificador reconhece este símbolo o processo é interrompido. Uma outra maneira seria informar ao decodificador quantos símbolos a mensagem possui, e, logo após serem decodificados tal número de símbolos, o processo é interrompido. O primeiro método mesmo gastando espaço “extra” com um símbolo especial é mais útil na implementação, pois,

(a)

(b)

Figura 1.2: Representações gráficas da codificação aritmética : (a) sem expansão, (b) com expansão após cada iteração.

não necessita do conhecimento prévio do comprimento da mensagem, o que é impossível em certos casos (por exemplo transmissões contínuas, para as quais os processos são executados ao mesmo instante em locais distintos). O espaço extra gasto deve-se ao fato da necessidade de alocação de um certo espaço de probabilidade a este símbolo, o qual só irá acontecer uma vez na mensagem. No próximo capítulo poderemos ver que o espaço gasto com este símbolo pode ser muito pequeno, sendo assim, ao longo deste trabalho empregaremos apenas o primeiro método. No nosso exemplo, o símbolo $\bullet$ é aquele usado para identificar o fim da mensagem, e assim, ao ser decodificado o processo é interrompido.

Vejam agora como se comporta a codificação aritmética em relação à quantidade de informação da mensagem. Para uma mensagem $M = x_1x_2\dots x_t$, o código obtido pela codificação aritmética é um número compreendido em um intervalo de tamanho $\prod_{i=1}^t p(x_i, i)$, logo, o número de bits necessários para a sua representação é no mínimo $-\lg \prod_{i=1}^t p(x_i, i) = -\sum_{i=1}^t \lg p(x_i, i) = I(M)$. Com isto, vemos que o código aritmético codifica a mensagem com o comprimento igual à sua quantidade de informação.

No exemplo acima, a quantidade de informação da mensagem $eeio\bullet$ é $-\lg 0.3 - \lg 0.3 - \lg 0.1 - \lg 0.1 - \lg 0.1 = -\lg 0.00009 \approx 13$ bits, o intervalo final resultante é $[0.59101, 0.5911)$ e para representarmos um número neste intervalo (por exemplo, $r = 0.59101$) necessitamos de $-\lg(0.59101 - 0.5911) = -\lg 0.00009 \approx 13$ bits. Para uma comparação com o código de Huffman, temos que o código de Huffman para M é 10 10 110 1110 1111, totalizando 15 bits. O código de Huffman foi pior neste caso, pois ele não conseguiu codificar alguns símbolos com a sua real quantidade de informação. Por exemplo, o símbolo e é codificado com uma palavra-código de 2 bits pelo código de Huffman quando deveria ser codificado com uma palavra-código menor ($\lg 0.3 \approx 1.73$ bits). O mesmo ocorre com o símbolo o e o símbolo $\bullet$. Devemos salientar também que a mensagem usada no exemplo não está de acordo com a distribuição de probabilidades usada, podemos ver, por exemplo, que o símbolo a não aparece na mensagem, porém possui probabilidade alta.

O número de bits para codificar a mensagem pode variar de acordo com o modelo usado. Se ao invés da tabela 1.1 usássemos a seguinte distribuição fixa de probabilidades: $\{a(0), e(0.4), i(0.2), o(0.2), \bullet(0.2)\}$ necessitaríamos de apenas 9 bits. Mais adiante, estudaremos modelos mais sofisticados, que obtêm melhores resultados.

Nas próximas seções estudaremos duas implementações da codificação aritmética, e uma análise completa dos problemas devidos à sua implementação. A primeira implementação [8] requer o uso de multiplicações e divisões para o ajuste dos intervalos de código, diferentemente da segunda [15] a qual dispensa o uso de tais operações, tornando-se facilmente adaptável a uma implementação em hardware.

1.5 Implementando o Código Aritmético

O algoritmo básico para a implementação do código aritmético pode ser obtido de maneira muito simples, baseando-se na descrição dada na seção anterior. A figura 1.3 mostra um algoritmo genérico que implementa, de maneira simples, a codificação e decodificação aritmética.

Os m símbolos do alfabeto A são ordenados segundo alguma função o de ordenação, entre 1 e m , sendo que, de agora em diante, ao nos referirmos a algum símbolo estaremos nos referindo ao seu índice na ordenação o . Seja $Prob_cum$, uma lista que mantém as probabilidades acumulativas P de cada símbolo s , tal que $Prob_cum[s]$ decresce conforme s aumenta, logo $Prob_cum[0] = 1$ e a probabilidade do símbolo s vale $Prob_cum[s - 1] - Prob_cum[s]$. O intervalo ao final de cada iteração é $[ini, fim)$ e, tanto para a codificação quanto para a decodificação, ele deve ser inicializado com $[0, 1)$.

O algoritmo da Figura 1.3 é muito genérico, e existem inúmeros fatores que atrapalham na sua implementação, tanto na codificação como na decodificação :

- O código (*valor*) não é obtido antes de toda mensagem ser processada, nem o decodificador começa a decodificação antes de receber todo o código (*valor*).

Solução: *Codificação e decodificação incrementais.*

- A precisão aritmética necessária para representar o intervalo cresce com o comprimento da mensagem.

Solução: Uso de *precisão aritmética finita*. A operação incremental auxilia no uso de precisão finita, porém, deve-se tratar cuidadosamente os casos de *overflow* e *underflow*.

- A decodificação pode gastar muito tempo até encontrar o próximo símbolo a ser decodificado.

```

/* ALGORITMO PARA CODIFICAÇÃO ARITMÉTICA */

/* O algoritmo é executado para todos os símbolos repetidamente */
/* Quando o símbolo de fim de mensagem é codificado um valor */
/* entre  $[ini, fim)$  é enviado. */

codifica(simb, Prob_cum);
    tam = fim - ini;
    fim = ini + tam * Prob_cum[simb - 1];
    ini = ini + tam * Prob_cum[simb];

/* ALGORITMO PARA DECODIFICAÇÃO ARITMÉTICA */

/* O algoritmo é executado até decodificar o símbolo de fim de mensagem */
/* valor é o código recebido. */

decodifica(Prob_cum);
    tam = fim - ini;
    procura símbolo s tal que
         $Prob\_cum[s] \leq (valor - ini) / tam < Prob\_cum[s - 1]$ 
    fim = ini + tam * Prob_cum[s - 1];
    ini = ini + tam * Prob_cum[s];
    retorna s

```

Figura 1.3: Algoritmos genéricos para codificação e decodificação aritmética.

Solução: a representação do *modelo* deve ser feita de maneira a minimizar esta pesquisa. Modelos adaptativos devem ser organizados de tal maneira que a operação de consultar e atualizar as probabilidades acumulativas seja minimizada.

1.5.1 Codificação e decodificação incrementais

Codificação e decodificação incrementais é um mecanismo utilizado com o propósito de ser possível iniciar a construção do código e posterior decodificação sem a necessidade de conhecer todo o código final previamente.

Vejamos o exemplo dado na seção anterior. Após receber o símbolo *i* o intervalo resultante é $[0.583, 0.592)$, logo, o código resultante é um número real da forma $0.5\dots$. Da mesma forma, após o símbolo *o* o código resultante é um número da forma $0.59\dots$, e assim por diante. A partir desta observação o codificador aritmético pode iniciar sua tarefa no momento em que um dígito estiver já definido, do mesmo modo o decodificador pode iniciar seu processo da maneira incremental. No exemplo, quando o decodificador recebe a parte do código igual a 0.59 , ele já sabe que a palavra *ee* é a única que contém o número 0.59 em seu intervalo resultante, logo já pode decodificá-la.

O processo incremental também faz com que não seja necessária precisão aritmética infinita, pois, quando um dígito é codificado ele não terá mais influência em nenhum outro cálculo no processo. No exemplo, se 0.59 do intervalo $[0.5902, 0.5911)$ é codificado, o intervalo pode ser trocado pelo intervalo $[0.0002, 0.0011)$ e ainda mais por $[0.02, 0.11)$, sem afetar os futuros cálculos, ou seja, o código final obtido com uso do processo incremental será o mesmo do código obtido sem o uso do processo incremental. Logo, o processo incremental diminui a precisão aritmética necessária ao processo, pois, como vimos no exemplo, ao passar do intervalo $[0.5902, 0.5911)$ para o intervalo $[0.02, 0.11)$ a precisão cai de 4 para 2 dígitos.

Proposição 2 *Se em um passo do algoritmo o intervalo resultante $[ini, fim)$ é do tipo $[0.w_1w_2\dots w_k i_1\dots i_n, 0.w_1w_2\dots w_k f_1\dots f_n)$; onde w_j, i_l e $f_l, 1 \leq j \leq k$ e $1 \leq l \leq n$ são dígitos segundo alguma base b , então ele pode ser substituído por $[0.i_1\dots i_n, 0.f_1\dots f_n)$ sem afetar os cálculos futuros.*

Prova: Observando a figura 1.3, notamos que o único fator que afeta o cálculo do novo intervalo é o tamanho *tam* do intervalo inicial do passo, que vale:

$$tam = fim - ini = [0.\underbrace{0\dots 0}_k \underbrace{(f_1\dots f_n) - (i_1\dots i_n)}_{t>0}]_b = [0.t]_b * b^{-k}.$$

E para o cálculo do novo intervalo a quantia somada ao antigo valor de *ini* vale:

$([0.t]_b * b^{-k}) * P = ([0.t]_b * P) * b^{-k}$, onde $0 \leq P \leq 1$ é uma probabilidade acumulativa de algum símbolo.

Logo, a quantia somada é, a menos de um deslocamento de k dígitos, igual ao deslocamento sofrido pela substituição. Sendo assim, os cálculos futuros terão resultados equivalentes, a menos dos deslocamentos efetuados. CQD.

Vejamos, agora, o comportamento da codificação e decodificação vista até o momento a partir de uma implementação. A primeira implementação que iremos estudar foi proposta em 1987 por Witten, Neal e Cleary[8], e implementada por nós. Nesta implementação, o intervalo será representado por um intervalo inteiro $[ini, fim)$, que inicialmente possui tamanho $Topo + 1$; isto é, inicialmente $ini = 0$ e $fim = Topo$. Para o estudo desta implementação, os valores de *ini* e *fim* serão representados na base binária, e podemos associá-los a números fracionários que sofreram um deslocamento igual à precisão usada, a qual é igual ao número de bits necessários para representar o intervalo, ou seja, $\lg(Topo + 1)$. O valor *default* para a precisão será de 16, logo, $Topo = 2^{16} - 1 = 65535$. Na verdade, a implementação faz com que o intervalo seja fechado tanto na parte inferior como na superior, sendo que para isto, *fim* deve assumir o valor $Topo + [0.11111\dots]_2$ (representação binária do número 0.9999...). Para obtermos esta notação é necessário que, sempre que houver codificação de 1 bit, e posterior deslocamento do intervalo para a esquerda, bits “0” sejam deslocados para a parte inferior de *ini* e bits “1”, para a parte inferior de *fim*. O método de manter o intervalo fechado em ambas as extremidades visa, simplesmente, facilitar a implementação, porém, uma implementação sem esta restrição é igualmente possível de ser obtida². As probabilidades de cada símbolo são representadas por valores inteiros normalizados, isto é, existe uma função *freq* que associa a cada símbolo s um valor inteiro $freq(s)$, que é denominada por frequência do símbolo s . Com isto, a probabilidade de s será igual a $freq(s)/norma$, onde $norma = \sum_{k \in A} freq(k)$. Além disso, é mantida uma função *cum_freq* associando a cada símbolo s um valor inteiro $cum_freq(s)$. Esta função é de-

²Esta implementação necessitaria de um número maior de comparações ao longo do processo.

nominada de frequência acumulativa de s , sendo a probabilidade acumulativa de s igual a $cum_freq(s)/norma$. Não nos preocupamos em discriminar o instante i das funções $freq$ e cum_freq , assumiremos, então, que estas funções são fixas, ou seja, não se alteram durante o processo; ou são atualizadas conforme cada símbolo é processado.

Em cada etapa do processo, as frequências acumulativas são projetadas no intervalo atual $[ini, fim]$, e ao receber um símbolo, o codificador recalcula novos ini e fim para a próxima etapa. Após este cálculo, os valores ini e fim podem estar muito próximos, e possuírem bits significativos iguais. Se isto ocorrer o processo incremental deve ser executado, ou seja, estes bits devem ser colocados no código final e o intervalo re-dimensionado. Uma maneira de executar este procedimento é

```

enquanto  $fim < Meio$  ou  $ini \geq Meio$ 
  se  $fim < Meio$ 
     $saibit(0)$ ;
  senão
     $saibit(1)$ ;
   $ini = ini \ll 1$ ;
   $fim = fim \ll 1 + 1$ ;

```

onde $saibit(b)$ é uma função que coloca o bit b no código final, e $Meio$ é a metade do tamanho inicial, ou seja, $Meio = (Topo + 1)/2$ (o valor *default* é $Meio = 2^{15} = 32768$). O procedimento acima faz com que sempre tenhamos $ini < Meio \leq fim$. Vale a pena notar que bits 0 são deslocados para ini , enquanto bits 1 são deslocados para fim à medida que o intervalo é re-dimensionado.

Para a decodificação temos uma variável *valor* que contém a parte do código lido até o momento. Ao iniciar o processo, *valor* recebe os primeiros bits do código final com os quais é possível determinar o próximo símbolo³. Ao reconhecer o símbolo, o novo intervalo $[ini, fim]$ é recalculado e re-dimensionado da mesma forma que o codificador. Os deslocamentos em fim e ini no re-dimensionamento do intervalo também se aplicam a *valor*,

³Na próxima seção ficará claro que o decodificador consegue reconhecer o próximo símbolo a partir de *valor*.

sendo que este recebe, como bit menos significativo, o próximo bit do código final a ser decodificado. Pelo procedimento abaixo, verificamos como este processo é executado.

```

enquanto fim < Meio ou ini ≥ Meio
    ini = ini << 1;
    fim = fim << 1 + 1;
    valor = valor << 1 + entrabit();

```

onde *entrabit*() retorna o próximo bit do código final a ser decodificado.

1.5.2 O Problema da Decodificação

Veremos agora como o decodificador consegue determinar o próximo símbolo a partir de *valor*. Inicialmente o decodificador projeta a variável *valor* no espaço das frequências acumulativas, daí em diante, a tarefa é apenas a de percorrer *cum_freq* e verificar em qual intervalo *valor* se encaixa. O procedimento abaixo executa este processo, além de recalculer o novo intervalo [*ini*, *fim*].

```

tam = fim - ini + 1;
cum1 = ((valor - ini + 1) * norma - 1) / tam;
s = procura_simbolo(cum1);
fim' = ini + (tam * (cum_freq[s] + freq[s])) / norma - 1;
ini' = ini + (tam * (cum_freq[s])) / norma;

```

onde *cum1* é a projeção de *valor* no espaço das frequências acumulativas. A função *procura_simbolo* percorre *cum_freq* tentando identificar algum símbolo *s* tal que

$$cum_freq[s] \leq \left\lfloor \frac{(valor - ini + 1) * norma - 1}{fim - ini + 1} \right\rfloor < cum_freq[s + 1]. \quad (1.1)$$

Para ser possível o processo de decodificação devemos manter a invariante de que *valor* sempre está compreendido nos intervalos $[ini, fim]$ calculados em cada passo do processo pelo procedimento acima. Para demonstrar que esta invariante é mantida ao longo da decodificação, basta notarmos que se *valor* estiver compreendido no intervalo $[ini, fim]$ no início do procedimento então ele estará no fim do mesmo, ou seja, no intervalo $[ini', fim']$. Reformularemos a inequação 1.1 por

$$cum_freq[s] \leq \frac{(valor - ini + 1) * norma - 1}{tam} - \varepsilon \leq cum_freq[s - 1] - 1, \quad (1.2)$$

onde $0 \leq \varepsilon \leq (tam - 1)/tam$. A última inequação de 1.2 é obtida por *cum_freq* ser inteiro. Devemos mostrar agora que $ini' \leq value \leq fim'$.

$$\begin{aligned} (I) \quad ini' &\equiv ini + \left\lfloor \frac{tam * cum_freq[s]}{norma} \right\rfloor \\ &\stackrel{\text{de 1.2}}{\leq} ini + \frac{tam}{norma} \left\lfloor \frac{(valor - ini + 1) * norma - 1}{tam} - \varepsilon \right\rfloor \\ &\leq valor + 1 - \frac{1}{norma} \end{aligned}$$

como *valor* e *ini'* são inteiros e *norma* > 0, temos que $ini' \leq valor$. Também vale que

$$\begin{aligned} (II) \quad fim' &\equiv ini + \left\lfloor \frac{tam * cum_freq[s - 1]}{norma} \right\rfloor - 1 \\ &\stackrel{\text{de 1.2}}{\geq} ini + \frac{tam}{norma} \left\lfloor \frac{(valor - ini + 1) * norma - 1}{tam} + 1 - \varepsilon \right\rfloor - 1 \end{aligned}$$

$$\begin{aligned}
&\geq \text{valor} + \frac{\text{tam}}{\text{norma}} \left[-\frac{1}{\text{tam}} + 1 - \frac{\text{tam} - 1}{\text{tam}} \right] \\
&= \text{valor}
\end{aligned}$$

Logo, tomando (I) e (II) temos que $\text{ini}' \leq \text{valor} \leq \text{fim}'$. Com isto provamos que vale a invariante. Devemos salientar, ainda, que a invariante é mantida mesmo quando ocorre o re-dimensionamento do intervalo, pois em *ini* entra bit 0 e em *fim* entra bit 1, sendo que, se *valor* está compreendido no novo intervalo antes do re-dimensionamento, estará ele no intervalo após o re-dimensionamento. Ante a validade desta invariante podemos dizer assim, que o processo de decodificação ocorre sem nenhum problema.

1.5.3 Tratamento de Overflow e Underflow

Existem ainda dois problemas a serem tratados na implementação da codificação aritmética, são eles: o *underflow* e o *overflow*. O *underflow* ocorre no caso em que o intervalo se torna tão pequeno que um símbolo, quando projetado no intervalo $[\text{ini}, \text{fim}]$, ocupa espaço nulo. Como a menor probabilidade que um símbolo pode ter, é $1/\text{norma}$, então, para o *underflow* ocorrer, basta que o valor da *norma* seja maior que o tamanho *tam* do intervalo: $\text{fim} - \text{ini} + 1$, pois, sendo este o caso, a probabilidade projetada do símbolo seria nula. Para evitar tal caso, o codificador deve tanto manter o intervalo sempre maior que o valor da *norma*, quanto, principalmente mantê-lo sempre maior que o “máximo” valor que a *norma* puder valer. Em nossa implementação este valor será denominado por *Max_norma*. Os casos nos quais as freqüências dos símbolos vão sendo alteradas, o valor da *norma* pode ultrapassar o máximo permitido. Existem vários mecanismos para evitar tais casos, um deles é dividir todas as freqüências relativas a cada símbolo por um certo valor sempre que *norma* ultrapassar o seu máximo. Este procedimento será visto no próximo capítulo.

Já vimos que, pelo processo incremental, quando *ini* e *fim* possuem o bit mais significativo idêntico, este é enviado para o código e o intervalo é duplicado. Existe um caso, porém, que o intervalo pode diminuir consideravelmente de tamanho, e não ser possível a execução do processo incremental. Por exemplo, suponha que ocorra a seguinte desigualdade

$$Quarto_1 \leq ini < Meio \leq fim < Quarto_3 \quad (1.3)$$

onde $Quarto_1$ é o 1º quarto e $Quarto_3$ é o 3º quarto do intervalo inicial de tamanho $Topo + 1$ (para $Topo = 65535$, então $Quarto_1 = \frac{Topo+1}{4} = 16384$, $Meio = 2 * \frac{Topo+1}{4} = 32768$ e $Quarto_3 = 3 * \frac{Topo+1}{4} = 49152$). Neste caso não é possível a execução do processo incremental, pois, os bits mais significativos de ini e fim são opostos, ainda mais os dois bits mais significativos de ambos são opostos, 10 para fim e 01 para ini . Se o próximo bit a ser codificado pelo processo incremental for 0 (quando $fim < Meio$), então o bit seguinte deverá ser oposto igual a 1, por tratar-se do caso em que $Quarto_1 \leq ini < fim < Meio$. Caso o próximo bit a ser codificado for 1 (quando $ini \geq Meio$), então o bit seguinte, também será oposto e neste caso igual a 0, vez que $Meio \leq ini < fim < Quarto_3$. A partir destas observações podemos expandir o intervalo, porém, sempre com a condição de que quando houver a codificação do próximo bit pelo processo incremental, um bit oposto adicional será codificado. A expansão do intervalo é feita deslocando-se ini e fim para a esquerda da mesma forma que o processo anterior, porém neste caso com a ressalva de ser mantido o bit mais significativo de ambos. Para este caso, o procedimento a ser executado é o seguinte :

$$\begin{aligned} ini &= (ini - Quarto_1) << 1; \\ fim &= (fim - Quarto_1) << 1 + 1; \end{aligned}$$

Uma vez realizado o procedimento acima, a desigualdade em 1.3 ainda pode se manter. Isto ocorre quando ini e fim estão tão próximos que não apenas os dois bits mais significativos de ambos são opostos, e sim, os $k + 1$ bits mais significativos, ou seja, os $k + 1$ bits mais significativos de fim valem $1 \underbrace{000...}_k$ e de ini valem $0 \underbrace{111...}_k$, logo se o próximo bit a ser enviado é b então os próximos k bits serão $\bar{b}^4$. Para isto, basta executar o procedimento de expansão anterior e incrementar uma variável global *bits_opostos* que guardará

⁴ $\bar{b}$ é o bit oposto a b , isto é, se $\bar{b} = 1 \Rightarrow b = 0$ e se $\bar{b} = 0 \Rightarrow b = 1$.

o número de bits opostos a serem enviados após o próximo. O procedimento final para a expansão do intervalo é:

```

enquanto  $Quarto\_1 \leq ini$  e  $fim < Quarto\_3$ 
     $ini = (ini - Quarto\_1) << 1;$ 
     $fim = (fim - Quarto\_1) << 1 + 1;$ 
     $bits\_opostos ++;$ 

```

Vale a pena observar que a variável *bits_opostos* deve ser inicializada com 0 e que o procedimento *saibit(b)* deve consultá-la e, após enviar um bit *b*, o mesmo deve enviar *bits_opostos* bits $\bar{b}$. Na figura 1.4 observamos o procedimento de expansão com 3 passos: em (a) o caso onde o primeiro bit a ser enviado vale 0, e em (b) o caso onde vale 1.

Assim como foi feito com o processo incremental, devemos mostrar que este re-dimensionamento não afeta cálculos futuros.

Proposição 3 *Se em um passo do algoritmo o intervalo resultante $[ini, fim)$ é do tipo : $[0.(w_1 - 1)\underbrace{\bar{b}...\bar{b}}_k i_1...i_n, 0.w_1 \underbrace{0...0}_k f_1...f_n)$; onde $w_1 > 0$, i_l e f_l , $1 \leq l \leq n$ são dígitos segundo alguma base $b \geq 2$ e $\bar{b} = b - 1$; então ele pode ser substituído por $[0.(w_1 - 1)i_1...i_n, 0.w_1 f_1...f_n)$ sem afetar os cálculos futuros.*

Prova: Basta observar que o único fator que afeta o cálculo do novo intervalo é o tamanho *tam* do intervalo inicial do passo, que vale:

$$tam = fim - ini = [0.\underbrace{0...0}_{k-1} \underbrace{(1f_1...f_n) - (i_1...i_n)}_{t>0}]_b = [0.t]_b * b^{-(k-1)}.$$

E para o cálculo do novo intervalo, a quantia somada ao antigo valor de *ini* vale:

$([0.t]_b * b^{-(k-1)}) * P = ([0.t]_b * P) * b^{-(k-1)}$, onde $0 \leq P \leq 1$ é a probabilidade acumulativa de algum símbolo.

Logo, a quantia somada é a menos de um deslocamento de $k - 1$ dígitos, igual ao deslocamento sofrido pela substituição. Sendo assim, os cálculos futuros terão os mesmos resultados a menos dos deslocamentos efetuados. CQD.

(a)

(b)

Figura 1.4: Re-dimensionamento do intervalo para prevenir o *underflow*.
 (a) caso em que o próximo bit a ser enviado é 0, (b) caso em que o próximo bit 1

Podemos agora, calcular o tamanho final do intervalo resultante. Temos que, após as expansões, uma das duas invariantes abaixo valem:

$$\begin{aligned} ini < Quarto_1 < Meio \leq fim \\ \text{ou} \\ ini < Meio < Quarto_3 \leq fim \end{aligned}$$

Logo, o tamanho do intervalo resultante é sempre maior do que um quarto do tamanho inicial, isto é, $\frac{Topo+1}{4}$. Sendo assim, para não ocorrer *underflow* é necessário que o máximo valor que *norma* pode assumir seja menor do que um quarto do tamanho total reservado ao intervalo, isto é:

$$Max_norma \leq \frac{Topo + 1}{4} - 1 \quad (1.4)$$

Se, por exemplo, o intervalo é representado com r bits, ou seja, $Topo = 2^r - 1$, então o maior valor para *Max_norma* deverá ser de $2^{r-2} - 1$.

Vejamos agora os casos em que pode ocorrer *overflow*. Sabemos que a variável *bits_opostos* é incrementada sempre que a desigualdade 1.3 ocorrer, a qual pode se dar muitas vezes antes de podermos executar o processo incremental, tornando a variável *bits_opostos* tão grande a ponto de provocar estouro em sua capacidade de armazenamento. Este é um caso raro, pois, para isto seria necessário que o código resultante possuísse uma quantidade muito grande de bits iguais consecutivos, mesmo assim, tal possibilidade existe e devemos nos preocupar com a mesma. Inúmeras maneiras de resolver este problema podem ser utilizadas, a mais usual é ao detectar a possibilidade de *overflow*, o codificador aloca espaço para uma extensão da variável *bits_opostos* e reinicializa a mesma com o valor adequado, e assim por diante, tornando a variável *bits_opostos* uma concatenação de variáveis. Outra maneira de resolver o problema seria o codificador supor a codificação de um símbolo redundante antes de ocorrer o *overflow*, o qual faria com que fosse enviado um bit (seguido de *bits_opostos* bits opostos) e a variável *bits_opostos* re-inicializada. Neste caso, o decodificador teria que ser capaz de detectar a ocorrência de um *overflow* no código e ignorar o próximo símbolo decodificado (redundante). Para isto, bastaria também manter uma variável *bits_opostos*, a qual é incrementada quando vale a desigualdade 1.3 e re-inicializada quando do processo incremental. Sendo assim, ao detectar *overflow* em *bits_opostos*

o decodificador ignora o próximo símbolo decodificado. Em nossa implementação não nos preocupamos com este caso de *overflow*, por ser um caso muito raro de acontecer.

Outro caso de *overflow* é o das multiplicações para o cálculo do novo intervalo. Estas multiplicações são do tipo $tam * cum_freq[s]$ e são sempre menores ou iguais à $tam * norma \leq tam * Max_norma \leq Topo * Max_norma \leq 2^{2r-2}$, portanto, para se evitar o *overflow* destas multiplicações, o *hardware* deve ser capaz de fazer operações com precisão aritmética de $2r-2$ bits. Como exemplo, se usarmos um computador com precisão de 32 bits, podemos tomar $r = 16$, com isto $Topo = 2^{16} - 1$ e $Max_norma = 2^{14} - 1$, e assim a inequação 1.4 é satisfeita, e a precisão requerida de 30 bits não ultrapassa a máxima disponível de 32 bits.

O decodificador, por sua vez, também deve executar os mesmos passos do codificador para prevenção de *underflow* sem a necessidade da variável *bits_opostos*⁵, já que sua operação é decodificar um símbolo e somente depois recalculando o novo intervalo. A expansão do intervalo, porém, deve se comportar da mesma maneira que para o codificador. O procedimento abaixo mostra como se dá a implementação desta expansão.

```

enquanto  $Quarto\_1 \leq ini$  e  $fim < Quarto\_3$ 
     $ini = (ini - Quarto\_1) << 1$ ;
     $fim = (fim - Quarto\_1) << 1 + 1$ ;
     $valor = (valor - Quarto\_1) << 1 + entrabit()$ ;

```

Finalizada a codificação, após a codificação do último símbolo, o codificador deve enviar bits necessários para assegurar que o código obtido esteja dentro do último intervalo resultante. Como ao final da expansão, ou o 1º ou o 3º quarto pertence ao intervalo resultante, então pode-se enviar 01 no primeiro caso, e 10 no segundo, fazendo com que o código obtido pertença ao intervalo resultante. O decodificador por sua vez irá receber mais bits ao final do código para completar a variável *valor*, porém, estes não serão utilizados, já que o último símbolo enviado pode ser determinado apenas pelos dois últimos bits enviados.

⁵A menos do caso já visto para se evitar o *overflow* de *bits_opostos*.

1.5.4 Eficiência do código aritmético

Vimos que a codificação aritmética com precisão infinita é um processo de codificação, cujo código possui comprimento igual a soma dos logaritmos dos inversos das probabilidades das instâncias, valor conhecido por *comprimento ideal* do código. Também vimos que para a implementação, devemos abrir mão de algumas condições necessárias para o código atingir este comprimento ideal. Verificaremos nesta seção as perdas decorrentes desta implementação. Basicamente 3 fatores na implementação fazem com que a codificação não atinja o seu comprimento ideal:

- Problema do final de mensagem.
- Uso de precisão aritmética finita.
- Re-cálculo de frequências quando a *norma* atinge o valor máximo.

Sabemos que a codificação aritmética necessita de um símbolo especial de fim de mensagem. Na verdade, qualquer modelo deve alocar um espaço de código para indicar a ocorrência de fim de mensagem, pois este é um evento que pode ocorrer com alguma probabilidade⁶. Podemos usar modelos que tratam este evento de maneira distinta, como veremos no próximo capítulo, porém, sempre usando espaço adicional para o mesmo. Em nossa implementação, o espaço reservado para este evento tende a ser pequeno, pois vai diminuindo conforme o processo é executado. No próximo capítulo, trataremos este evento de maneira “especial”, logo, não entraremos em detalhes sobre o espaço gasto com ele no momento. Vimos também que na implementação o codificador deve enviar 2 bits adicionais, para retirar a ambigüidade do código. Como todo o processo é feito com cadeias de 8 bits (*byte*), então o número máximo de bits gasto é 9 (No pior caso o primeiro bit é o último de uma cadeia, e o segundo é o primeiro da próxima), e assim gastaríamos mais 1 byte).

O uso de precisão aritmética finita, é levado em conta pelo arredondamento feito no cálculo do novo intervalo, onde simplesmente são ignoradas

⁶Podemos assumir que este evento não ocorre se o processo for contínuo.

as partes não-inteiras dos novos *ini* e *fim*. A perda decorrente deste fato é relativa ao número de bits alocados para o intervalo. Para um intervalo de 16 bits, ou seja, $Topo = 2^{16} - 1$, a perda é da ordem de 10^{-4} bits/símbolo⁷.

Vamos demonstrar que, do mesmo modo que o código de Huffman, a implementação vista atinge o comprimento ideal, para um modelo fixo onde a probabilidade de cada símbolo é do tipo 2^{-n} e $m = 2^n$ é o número de símbolos do alfabeto, e o símbolo de fim de mensagem faz parte do alfabeto.

Proposição 4 *Seja b o número de bits alocados para o intervalo do codificador, logo, $Topo = 2^b - 1$. Seja um modelo fixo que caracteriza uma fonte, a qual emite símbolos sobre um alfabeto A de $m = 2^n$ símbolos, onde a probabilidade de cada símbolo a_i vale $P(a_i) = 2^{-n}$. Logo, o comprimento do código gerado pela implementação é*

$$T = \sum_{i=1}^m q_i * \lg P(a_i), \text{ onde } q_i \text{ é o número de ocorrências do símbolo } a_i.$$

Prova: Inicialmente mostraremos que em qualquer iteração da implementação o intervalo resultante é $[0, 2^b - 1]$. Para isto, assumiremos que $norma = 2^n$, $freq[a_i] = 1, \forall i = 1, \dots, m$; e $cum_freq[a_i] = i, \forall i = 1, \dots, m$.

Para esta demonstração usaremos indução na iteração k da implementação.

Base : Para $k = 0$, é satisfeita a condição pois o intervalo inicial vale $[0, 2^b - 1]$.

H.I. : Vale para iterações menores que $k > 0$.

Tese : Mostraremos que vale para iterações $k > 0$.

Pela **H.I.**, na iteração $k - 1$ o intervalo resultante é $[0, 2^b - 1]$. Supondo que o símbolo a_j é o símbolo a ser codificado na iteração k , temos que :

$$\begin{aligned} fim' &= ini + (fim - ini + 1) * (cum_freq[a_j] + freq[a_j]) / norma - 1 \\ &= 2^b * (j + 1) / 2^n - 1 = 2^{b-n} * (j + 1) - 1 \\ ini' &= ini + (fim - ini + 1) * cum_freq[a_j] / norma \\ &= 2^b * j / 2^n = 2^{b-n} * j \end{aligned}$$

O tamanho do novo intervalo $[ini', fim']$ vale 2^{b-n} , como ini' é um múltiplo de 2^{b-n} , então, ini' é da forma $[j]_2 \underbrace{[000..0]}_{b-n}_2$, e como fim' é da forma $ini' +$

⁷Este valor foi obtido em [8], porém, não foi demonstrado.

$\underbrace{[111\dots 1]}_{b-n}_2$, então, os n bits mais significativos de fim' e ini' são iguais. Logo pelo processo incremental são codificados n bits pelo codificador, e ambos fim' e ini' são deslocados n posições para esquerda tornando o intervalo resultante $[0, 2^b - 1]$.

A partir disto temos que a cada iteração o codificador codifica n bits e o intervalo resultante vale $[0, 2^b - 1]$, logo, o tamanho da saída é:

$$n * \sum_{a_i} q_i = \sum_{a_i} q_i * \lg P(a_i) = T. \text{ CQD.}$$

O recálculo das frequências quando *norma* atinge o valor máximo, só deve ser levado em conta por modelos adaptativos, pois, apenas este tipo de modelo varia a frequência relativa de cada símbolo. Na maioria das vezes, este recálculo acarreta um ganho de compressão. Isto se deve ao fato, de que ao recalcular frequências eventos mais recentes gerados pela fonte terão mais ênfase do que os gerados inicialmente.

O tempo gasto com a codificação aritmética é linear no comprimento da mensagem, pois um símbolo é codificado por vez sem haver consulta a símbolos anteriores. Para cada símbolo o número máximo de passos que o algoritmo executa é $\lg(1/Max_norma) = (r - 2)$ bits, onde r é o número de bits usados para representar o intervalo, isto ocorre quando o intervalo se torna unitário e devemos executar $(r - 2)$ expansões. Logo, para mensagens de comprimento t , o tempo gasto é da ordem de $O(tr)$. Veremos no próximo capítulo que existe o tempo necessário para o cálculo das probabilidades em cada iteração do processo, porém, como neste capítulo estamos apenas interessados na codificação aritmética, fizemos a suposição que este tempo é de $O(1)$.

1.6 Uma segunda implementação

Nesta seção veremos uma nova implementação estudada em 1989 por Rissanen e Mohiuddin[15], a qual não requer o uso de divisões nem de multiplicações. Uma implementação parecida para o alfabeto binário já havia sido estudada anteriormente por Langdon e Rissanen[12] em 1982, porém, este novo método se aplica, também, para o caso de alfabetos não binários. Por não usar multiplicações nem divisões, este método pode ser utilizado sem grandes problemas numa implementação em hardware.

A implementação de Langdon e Rissanen[12] para alfabetos binários é baseada em substituir a probabilidade do símbolo menos provável, pela sua melhor aproximação a uma potência inteira de $1/2$. Feito isso, as multiplicações para encontrar o novo intervalo podem ser substituídas por uma operação de deslocamento. Este mesmo procedimento não pode ser aplicado para alfabetos com mais de 2 símbolos pela razão de que as probabilidades destes símbolos não podem, ao mesmo tempo, serem todas aproximadas a potências de $1/2$.

Vimos, na seção anterior, que o codificador aritmético constrói um código que representa um número real pertencente ao intervalo relativo à probabilidade acumulativa da mensagem s a ser codificada. Esta pode ser obtida da mesma forma que a probabilidade acumulativa de uma instância, ou seja, pela soma das probabilidades de mensagens *menores* do que s . Dizemos que uma mensagem s' é menor que uma mensagem s , se s e s' têm o mesmo comprimento, e o primeiro índice k onde $s_k \neq s'_k$ implica que no instante k , o símbolo s_k tem índice menor que s'_k na ordenação feita sobre o alfabeto A . Como usamos precisão finita, o codificador necessita aproximar as probabilidades para a precisão requerida. Nesta seção, para simplicidade de escrita, denominaremos por $A(s)$ a aproximação da probabilidade $p(s)$ da mensagem s , que, vista como na seção anterior, é o tamanho do intervalo final $[ini, fim]$. Vamos supor, também, que para cada etapa do processo de codificação os $(m + 1)$ símbolos recebem uma ordem de 0 à m , e denominaremos o símbolo pelo seu índice nesta ordenação. Sendo assim, se $C(s)$ é o código relativo à mensagem s , então, o código da mensagem si pode ser obtido por

$$\begin{aligned} C(s0) &= C(s) \\ C(si) &= C(s0) + A(s0) + \dots + A(s(i-1)), i > 1 \end{aligned} \quad (1.5)$$

onde $s(i-1)$ é a mensagem s seguida do símbolo $(i-1)$. As condições iniciais são $C(\lambda) = 0$ e $A(\lambda) = 1$, onde λ é a mensagem vazia. Devemos, ainda, levar em conta a seguinte recursão:

$$A(s) = A(s0) + \dots + A(sm) \quad (1.6)$$

a qual garante a decodificação [11], e que, sendo inicializado com 1, faz com que $A(s)$ defina uma verdadeira fonte de informação. Estudaremos, mais

adiante, como $A(s)$ é construído recursivamente. Podemos verificar que $C(si)$ equivale à probabilidade acumulativa do símbolo i após a projeção no intervalo resultante para s .

As operações aritméticas usadas nas recursões 1.5 e 1.6, serão executadas usando dois registradores de tamanho fixo w^8 , sendo um para $C(s)$ e o outro para $A(s)$. Do mesmo modo que a última implementação, usaremos o processo incremental a medida que tentarmos manter $A(s)$ dentro de um intervalo pré-definido, e sempre que o valor $A(s)$ estiver abaixo do limite inferior deste intervalo devemos normalizá-lo. Isto será possível se introduzirmos a normalização $a(s) = 2^{L(s)}A(s)$, onde $L(s)$ é um inteiro não negativo determinado pela condição de $a(s)$ ser um número binário de w bits mantido sempre dentro de um intervalo fixo. Analisaremos, nas próximas seções, o mecanismo usado para a determinação deste intervalo de normalização, e assumiremos por ora o seguinte intervalo:

$$0.75 \leq a(s) < 1.5 \quad (1.7)$$

A operação de normalização é fácil de ser realizada, pois são necessárias apenas operações de deslocamento para a mesma. Com esta nova notação a recursão em 1.6 pode ser vista como

$$a(s) = \hat{a}(s0) + \dots + \hat{a}(sm) \quad (1.8)$$

onde $\hat{a}(si) = 2^{L(s)}A(si)$. À medida que s aumenta em comprimento, por 1.6, $A(s)$ diminui e $L(s)$ aumenta, sendo que as quantidades adicionadas ao código de acordo com 1.5 se movem para a direita. Do mesmo modo, o código $C(s)$ é deslocado para a esquerda relativamente ao registrador fixo onde as adições ocorrem, indicando assim o uso do processo incremental já visto.

Para nossas implementações as probabilidades de cada símbolo i , a cada passo, estarão de acordo com uma frequência n_i , determinada pelo modelo, dividida por $n = \sum_i n_i^9$. Também vamos assumir que em cada passo o símbolo com o maior índice (índice m) é aquele com maior probabilidade, logo, vale:

⁸Geralmente será tomado $w = 16$ ou 12 .

⁹Podemos verificar que n_i equivale à $freq(i)$ e n à $norma$, conforme definição na primeira implementação.

$$n_m \geq n_i, \forall i = 0, \dots, m-1. \quad (1.9)$$

1.6.1 Implementação para um alfabeto binário

A implementação para um alfabeto binário é mais simples, assim a descreveremos primeiro. Vamos assumir que no passo atual x seja o símbolo com menor probabilidade, logo $n_x \leq n/2$, e $\bar{x}$ o símbolo com maior probabilidade. Inicialmente iremos normalizar n para o intervalo descrito em 1.7 obtendo $\hat{n}$. Representaremos n com w bits, onde seu primeiro bit é a sua parte inteira e os $w-1$ bits restantes, sua parte fracionária. Para a normalização devemos deslocar n de $k(n)$ posições para a esquerda, para isto basta verificar os dois bits mais significativos de n ¹⁰. Se o segundo bit mais significativos de n é 0 então $k(n) = w - l(n)$, caso contrário, $k(n) = w - l(n) - 1$, onde $l(n)$ é o número de bits significativos de n . Por exemplo, seja $w = 8$. Se $n = 00010110$, então $\hat{n} = 1.0110000$, porém, se $n = 00011110$ então $\hat{n} = 0.1111000$. Nos dois casos $l(n) = 5$, no primeiro $k(n) = 8 - 5 = 3$ e no segundo $k(n) = 8 - 5 - 1 = 2$. Com isto, temos que $\hat{n} = 2^{k(n)}n$. O mesmo deslocamento executado para n é feito para cada n_i , obtendo $\hat{n}_i = 2^{k(n)}n_i$. No caso em que os símbolos são representados por suas probabilidades p_i e não por suas frequências n_i , devemos olhar cada probabilidade representada com $w-1$ dígitos fracionários e assumir $n = \hat{n} = 1$, e $\hat{n}_i = p_i$. Como temos que $n = n_x + n_{\bar{x}}$, então $\hat{n} = \hat{n}_x + \hat{n}_{\bar{x}}$.

A recursão a seguir será usada para satisfazer a decomposição em 1.8 quando $m = 1$

$$\hat{a}(sx) = a(s)p(x | s), \hat{a}(s\bar{x}) = a(s) - a(s)p(x | s), \quad (1.10)$$

onde $p(x | s)$ é a probabilidade de ocorrer o símbolo, x dado que ocorreu a mensagem s ¹¹. O objetivo é reservar para $a(si)$ um espaço em $a(s)$ proporcional a probabilidade de i dado que ocorreu s . Como não desejamos usar

¹⁰O bit mais significativos é o primeiro bit diferente de 0 percorrendo o número da esquerda para a direita.

¹¹Veremos no próximo capítulo como pode ser calculada esta probabilidade.

multiplicações nesta implementação, inicialmente devemos aproximar a probabilidade condicional $p(x | s)$ por $n_x/n = \hat{n}_x/\hat{n}$, e então obter a partir de 1.10 a seguinte equação:

$$\hat{a}(sx) = a(s) \frac{\hat{n}_x}{\hat{n}} = \hat{n}_x \frac{a(s)}{\hat{n}}. \quad (1.11)$$

Pela equação 1.7 $a(s)$ e $\hat{n}$ estão normalizados dentro do mesmo intervalo, logo, $a(s)/\hat{n}$ é da ordem de 1. Com isto, a equação 1.11 é aproximada por

$$\hat{a}(sx) = \hat{n}_x, \hat{a}(s\bar{x}) = a(s) - \hat{n}_x, \quad (1.12)$$

e $\hat{a}(si)$ é renormalizado para o intervalo em 1.7.

Vamos ilustrar o processo com um exemplo. Suponha $n = 1101$ e $n_0 = 0010$ nas representações binárias. Após as normalizações temos: $\hat{n} = .1101$ e $\hat{n}_0 = .0010$. Seja $a(s) = 1.001$, após a recursão 1.12 temos que $\hat{a}(s0) = .0010$ e $\hat{a}(s1) = 1.000$. Mesmo com $\hat{a}(s0)/a(s) = 0.0010/1.001 = 1/9$ e $\hat{n}_0/\hat{n} = 0.0010/0.1101 = 2/13$, diferindo por volta de 50 por cento, o efeito no comprimento do código mostra-se pequeno. Estudaremos este resultado nas próximas seções.

A figura 1.5 descreve o algoritmo para a codificação. Ele usa dois registradores C e A com tamanho w bits. Interpretaremos o valor de C como um número binário fracionário com todos seus dígitos fracionários, e o valor de A também como um número binário fracionário, onde seu dígito mais significativo é sua parte inteira e os demais sua parte fracionária. O código $C(s)$ será construído a partir dos bits que são deslocados para fora de C juntos com os bits do próprio C . Inicialmente $C = 0 = \underbrace{00\dots00}_{w \text{ bits}}$ e $A = 1 = 1 \underbrace{00\dots00}_{(w-1) \text{ bits}}$.

Na figura 1.5 $LimInf = 0.75$ e $LimSup = 1.5$; a função *envia_bitC()* coloca na saída o bit mais significativo de C , e x é o símbolo de menor probabilidade. O procedimento acima é executado cada vez que é recebido um símbolo x , e ao final quando não há mais símbolos a processar, o registrador C é enviado para a saída.

/* ALGORITMO para CODIFICAÇÃO ARITMÉTICA */

/ O algoritmo é executado para todos os símbolos repetidamente */*
/ Quando não há mais símbolos para codificar C é enviado */*

```

CodBin( $y$ )
  se  $y = x$ 
     $A = \hat{n}_x$ ;
  senão
     $C+ = \hat{n}_x$ ;
     $A- = \hat{n}_x$ ;
  enquanto  $A < LimInf$  ou  $A \geq LimSup$ 
     $A <<= 1$ ;
    envia_bit $C()$ ;
     $C <<= 1$ ;

```

/* ALGORITMO para DECODIFICAÇÃO ARITMÉTICA */

/ O algoritmo é executado até que não haja mais bits para ler */*

```

DecodBin()
   $T = C - \hat{n}_x$ ;
  se  $T < 0$ 
     $simb = x$ ;
     $A = \hat{n}_x$ ;
  senão
     $simb = \bar{x}$ ;
     $C = T$ ;
     $A = A - \hat{n}_x$ ;
  enquanto  $A < LimInf$  ou  $A \geq LimSup$ 
     $A <<= 1$ ;
     $C = C << 1 + \text{le\_bit}()$ ;
  retorna  $simb$ 

```

Figura 1.5: Algoritmos para codificação e decodificação aritmética para um alfabeto binário, sem uso de multiplicações e/ou divisões.

A medida que C é incrementado pode ocorrer *overflow*. Langdon e Rissanen[12] usam uma rotina especial denominada de *bit-stuffing* para resolver o problema. No sentido de reduzir a frequência do problema de *overflow*, um registrador de tamanho w_c , que é colocado à esquerda de C , pode ser usado, resolvendo assim o problema para um número de *overflows* menores que w_c . Este mesmo registrador deve estar presente na decodificação.

Para ilustrar o algoritmo veremos um exemplo : seja $w = 4$, e um modelo fixo onde $n = 1000$, $n_0 = 0010$, $n_1 = 0110$, que normalizados viram $\hat{n} = 1.000$, $\hat{n}_0 = .0010$, $\hat{n}_1 = .0110$. Seja a mensagem a ser codificada igual à 0111010. A tabela 1.2 mostra os valores de C e A após a codificação de cada símbolo.

A figura 1.5 mostra a decodificação. Ela usa os mesmos registradores da codificação com idênticas interpretações, e o seu funcionamento é de maneira inversa ao do codificador. O registrador C é inicializado com os primeiros w bits lidos, e o registrador A com 1. Para ilustrar a decodificação usaremos os mesmos parâmetros utilizados no exemplo anterior. A tabela 1.3 mostra os valores de C e A após a decodificação de cada símbolo.

<i>Símbolo recebido</i>	<i>Reg. C</i>	<i>Reg. A</i>
início	0000	1000
0	00 0000	1000
1	00 0100	0110
1	001 0000	1000
1	001 0100	0110
0	00101 0000	1000
1	00101 0100	0110
0	0010101 0000	1000

Tabela 1.2: Exemplo de codificação para alfabeto binário

1.6.2 Implementando para um alfabeto não-binário

A principal idéia na normalização de $a(s)$, no caso de alfabetos binários, é fazer com que o número total das frequências $\hat{n}$ e $a(s)$ fiquem o mais próximo

<i>Símbolo decodificado</i>	<i>Reg. C</i>	<i>Reg. A</i>
início	0010	1000
0	1010	1000
1	0110	0110
1	0101	1000
1	0001	0110
0	0100	1000
1	0000	0110
0	0000	1000

Tabela 1.3: Exemplo de decodificação para alfabeto binário

possível, tornando viável, assim, a não necessidade do uso tanto de multiplicações quanto de divisões. Para isto, aproxima-se $a(si)$ por $\hat{n}_x$ ou $a(s) - \hat{n}_x$. Esta mesma idéia será aplicada para o caso de alfabetos com mais de dois símbolos. Inicialmente, devemos assumir que o símbolo de maior probabilidade para um alfabeto com m símbolos deve ser sempre o último símbolo da ordenação, o que é feito com o objetivo de reduzir erros de aproximação. Os outros símbolos, por sua vez, podem estar em qualquer ordem. Como no caso binário, $k(n)$ é o deslocamento de n para o intervalo em 1.7. A normalização dos contadores nesta nova implementação é descrita por:

$$\begin{aligned}\hat{n} &= 2^{k(n)}n \\ \hat{n}_i &= 2^{k(n)}n_i \\ \hat{Cum}(i) &= \sum_{j < i} \hat{n}_j, \text{ onde } \hat{Cum}(0) = 0.\end{aligned}$$

O contador $\hat{Cum}(i)$ têm significado análogo à freqüência acumulativa vista na primeira implementação, sendo neste caso igual a soma dos contadores normalizados $\hat{n}_j$, para $j < i$. Se mais uma vez os parâmetros forem probabilidades com $w - 1$ fracionários ao invés de freqüências, então teremos que $n = \hat{n} = 1$ e $\hat{n}_i = p_i$. A atualização dos $a(s)$ pode ser feita da seguinte maneira: $\hat{a}(si) = \hat{n}_i$, para $i < m$, e $\hat{a}(sm) = a(s) - \hat{Cum}(m)$. Neste caso, porém, há a possibilidade de que $\hat{Cum}(m) > a(s)$, já que $0.75 \leq a(s) < 1.5$ e $0 < \hat{Cum}(m) < 1.5$. Com isto, $\hat{a}(sm)$ poderá ser negativo, e para este caso,

trocaremos o valor de $k(n)$ por $k(n) - 1$, mantendo desta maneira $\hat{a}(sm)$ sempre positivo¹².

A figura 1.6 descreve o algoritmo para codificação, ele usa dois registradores C e A com tamanho w bits. As suas interpretações e suas inicializações são as mesmas feitas para o caso binário. Os valores de $\hat{Cum}(i)$ e $\hat{n}_i$ são os parâmetros do algoritmo. O código será construído a partir do registrador C e os bits que são deslocados para fora dele. O problema do *overflow* pode ser resolvido da mesma maneira vista para o alfabeto binário. A tabela 1.4 ilustra um exemplo de codificação em um modelo fixo, onde: $w = 4, n = 1000, n_0 = 0010, n_1 = 0011, n_2 = 0011$. Após a normalização temos: $\hat{n} = 1.000, \hat{n}_0 = 0.010, \hat{n}_1 = 0.011, \hat{n}_2 = 0.011$ e as frequências acumulativas são : $\hat{Cum}(0) = 0.000, \hat{Cum}(1) = 0.010, \hat{Cum}(2) = 0.101$. A mensagem recebida é 100212.

O algoritmo para decodificação também aparece na figura 1.6. O registrador C é inicializado com os primeiros w bits lidos, enquanto que o registrador A com 1. As interpretações destes registradores além dos parâmetros usados são análogos à codificação. A tabela 1.5 ilustra um exemplo de decodificação de um modelo fixo com os mesmos parâmetros do exemplo anterior.

<i>Símbolo recebido</i>	<i>Reg. C</i>	<i>Reg. A</i>
início	0000	1000
1	0 1000	0110
0	010 0000	1000
0	01000 0000	1000
2	010001 0100	0110
1	0100011 0000	0110
2	0100011101 0000	1000

Tabela 1.4: Exemplo de codificação para alfabeto não-binário

¹²Com esta troca teremos $0 < \hat{Cum}(m) < 0.75$.

```

/* ALGORITMO para CODIFICAÇÃO ARITMÉTICA */
/* Alfabeto não-binário */

```

```

/* O algoritmo é executado para todos os símbolos repetidamente */
/* Quando não há mais símbolos para codificar  $C$  é enviado */

```

```

CodNBin(y)
  se  $\hat{Cum}(m) < A$        $k = 0$ ;
  senão                 $k = 1$ ;
  se  $y = m$ 
     $C+ = \hat{Cum}(m) >> k$ ;
     $A- = \hat{Cum}(m) >> k$ ;
  senão
     $C+ = \hat{Cum}(y) >> k$ ;
     $A = \hat{n}_y >> k$ ;
  enquanto  $A < LimInf$  ou  $A \geq LimSup$ 
     $A <<= 1$ ;
    envia_bitC();
     $C <<= 1$ ;

```

```

/* ALGORITMO para DECODIFICAÇÃO ARITMÉTICA */
/* Alfabeto não-binário */

```

```

/* O algoritmo é executado até que não haja mais bits para ler */

```

```

DecodNBin()
  se  $\hat{Cum}(m) < A$        $k = 0$ ;
  senão                 $k = 1$ ;
  /* Encontra o maior símbolo  $simb$  tal que  $\hat{Cum}(simb) \leq C$  */
   $simb = 0$ ;
  enquanto  $simb \leq m$  e  $\hat{Cum}(simb) \leq C$ ;
     $simb++$ ;
   $simb--$ ;
  se  $simb = m$ 
     $C- = \hat{Cum}(m) >> k$ ;
     $A- = \hat{Cum}(m) >> k$ ;
  senão
     $C- = \hat{Cum}(simb) >> k$ ;
     $A = \hat{n}_{simb} >> k$ ;
  enquanto  $A < LimInf$  ou  $A \geq LimSup$ 
     $A <<= 1$ ;
     $C = C << 1 + le\_bit()$ ;
  retorna  $simb$ 

```

Figura 1.6: Algoritmos para codificação e decodificação aritmética para um alfabeto não-binário, sem uso de multiplicações e/ou divisões.

<i>Símbolo decodificado</i>	<i>Reg. C</i>	<i>Reg. A</i>
início	0100	1000
1	0000	0110
0	0011	1000
0	1110	1000
2	1101	0110
1	1010	0110
2	0000	1000

Tabela 1.5: Exemplo de decodificação para alfabeto não-binário

1.6.3 Eficiência do código

Analisaremos agora, a eficiência deste tipo de codificação. Esta análise, inclusive os resultados finais, foi transcrita do trabalho de Rissanen e Mohiuddin [15].

Inicialmente, assumiremos que a entrada no instante $t + 1$ para o símbolo x_{t+1} , é dada através de sua probabilidade $p(x_{t+1} | s^t)$, onde s^t é a mensagem processada até o instante t . Denominaremos $p(x_{t+1} | s^t)$ simplesmente por p_{t+1} . O cálculo do novo $a(s)$ é dado por

$$a(s^t x_{t+1}) = \begin{cases} \frac{p_{t+1}}{k(t)} & \text{se } x_{t+1} \neq m \\ a(s^t) - \frac{(1-p_m)}{k(t)} & \text{se } x_{t+1} = m, \end{cases} \quad (1.13)$$

onde $k(t) = 1$, se $a(s^t) > (1 - p_m)$, e $k(t) = 2$, c.c.; m é o último símbolo da ordenação no instante $t + 1$, e p_m a sua probabilidade. Veremos, a seguir, que m deve possuir a maior probabilidade para conseguirmos maior compressão.

Vamos definir L_c como sendo o comprimento do código obtido, e L_o o comprimento ótimo para um código, após N símbolos codificados. Seja, também, l_c o comprimento médio por símbolo obtido e l_o o comprimento médio ótimo por símbolo. Assim temos

$$L_c = - \sum_{t=0}^{N-1} \lg \frac{a(s^t x_{t+1})}{a(s^t)}$$

$$= - \sum_{t=0}^{N-1} \lg \frac{p(x_{t+1} | s^t)}{k(t)a(s^t)} + CP_m$$

onde CP_m é a correção devida quando o símbolo m é executado, sendo definido por

$$CP_m = - \sum_{t|x_{t+1}=m} \lg \left(a(s^t) - \frac{(1-p_m)}{k(t)} \right) - \lg \frac{p_m}{k(t)}. \quad (1.14)$$

Logo, temos

$$\begin{aligned} L_c &= - \sum_{t=0}^{N-1} \lg p(x_{t+1} | s^t) + \sum_{t=0}^{N-1} \lg(k(t)a(s^t)) + CP_m \\ &= L_o + \sum_{t=0}^{N-1} \lg(k(t)a(s^t)) + CP_m. \end{aligned}$$

Dividindo ambos lados por N , obtemos a ineficiência δ por símbolo do código, dada por

$$\delta = l_c - l_o = \frac{1}{N} \sum_{t=0}^{N-1} \lg(k(t)a(s^t)) + \frac{1}{N} CP_m. \quad (1.15)$$

Para grandes valores de N , temos

$$\frac{1}{N} \sum_{t=0}^{N-1} \lg(k(t)a(s^t)) \cong E(\lg(k(t)a(s^t))) \quad (1.16)$$

onde $E(\cdot)$ é o valor esperado. Vamos assumir que $a(s^t)$ determina um processo estacionário, uniformemente distribuído em seu intervalo $[0.75, 1.5]$. Isto é,

$$p(a(s^t)) = p(a) = \begin{cases} \frac{1}{0.75} & , 0.75 \leq a < 1.5 \\ 0 & , c.c. \end{cases} \quad (1.17)$$

Sabendo que $k(t) = 1$ quando $a(s^t) > (1 - p_m)$, e $k(t) = 2$ c.c., o primeiro termo em 1.15 pode ser aproximado por

$$\begin{aligned} \frac{1}{N} \sum_{t=0}^{N-1} \lg k(t) a(s^t) &\cong \frac{1}{0.75} \left(\int_{0.75}^{1-p_m} \lg 2a \, da + \int_{1-p_m}^{1.5} \lg a \, da \right) \\ &= 0.142267 + \frac{(1 - p_m) - 0.75}{0.75}. \end{aligned} \quad (1.18)$$

E segundo termo em 1.15 por

$$\frac{1}{N} CP_m \cong -p_m \left(E \left(\lg \left(a(s^t) - \left(\frac{1 - p_m}{k(t)} \right) \right) \right) - E \left(\lg \frac{p_m}{k(t)} \right) \right). \quad (1.19)$$

Ainda com a mesma distribuição uniforme de $a(s^t)$, temos

$$\begin{aligned} \frac{1}{N} CP_m &\cong -\frac{p_m}{0.75} \left(\int_{0.75}^{1-p_m} \lg \left(a - \frac{(1 - p_m)}{2} \right) da + \int_{1-p_m}^{1.5} \lg(a - (1 - p_m)) da \right) \\ &\quad + p_m \lg p_m - p_m p(a(s^t) \leq 1 - p_m) \\ &= -\frac{p_m}{1.5} \left((1 - p_m) \lg \frac{(1 - p_m)}{2e} + (0.5 + p_m) \lg \frac{(1 + 2p_m)}{e} \right) + p_m \lg p_m \\ &\quad - p_m \frac{(1 - p_m) - 0.75}{0.75} \end{aligned} \quad (1.20)$$

onde e é a base do logaritmo natural. De 1.15, 1.18 e 1.20, obtemos δ por

$$\begin{aligned} \delta &= 0.142267 + (1 - p_m) \frac{(1 - p_m) - 0.75}{0.75} + p_m \lg p_m \\ &\quad - p_m \left(\frac{1 - p_m}{1.5} \lg \frac{1 - p_m}{2e} + \frac{0.5 + p_m}{1.5} \lg 1 + 2p_m e \right) \end{aligned} \quad (1.21)$$

e a eficiência da compressão é

$$\text{Eficiência} = \frac{l_o}{l_c} = \frac{l_o}{l_o + \delta}. \quad (1.22)$$

Na língua inglesa o símbolo mais freqüente costuma ser a letra t , com probabilidade igual à 0.0624. Também temos que o comprimento médio ótimo para cada símbolo para textos em inglês é de 4.7887¹³. Assumindo t como o último símbolo (símbolo m) para compressão de arquivos contendo textos em inglês temos $p_m = 0.0624$, e $l_o = 4.7887$ bits/símbolo. De 1.21 verificamos que δ vale 0.2556 bits/símbolo, e de 1.22 a eficiência é de 94.9 por cento. Isto coincide bem com os resultados experimentais da tabela 1.7 e qualquer desvio em relação aos resultados experimentais se deve à suposição de que $a(s^t)$ é uniformemente distribuída.

Na análise acima, supomos que $0.75 \leq (1 - p_m) < 1.5$, o que geralmente acontece. Para grandes valores de p_m , é possível que $(1 - p_m) < 0.75$, e a análise se torna mais simples, pois, deste modo, $k(t) = 1$ em 1.13.

Em 1.21, δ depende de p_m , quando p_m cresce, δ decresce, implicando em maior eficiência. Vê-se esta variação na tabela 1.6, na coluna $\delta(0.75)$. Sendo assim, recomendada-se que o símbolo de maior probabilidade seja o último.

Vamos agora analisar o intervalo em que $a(s)$ deve ser normalizado, o qual foi assumido até o momento como sendo $[0.75, 1.5)$. Seja o intervalo de normalização igual à $[u, 2u)$, e $u \leq (1 - p_m) < 2u$. Podemos obter uma expressão geral para δ em termos de u como sendo

$$\begin{aligned} \delta(u) = & (1 - p_m) \frac{(1 - p_m) - u}{u} + 2 + \lg \frac{u}{e} + p_m \lg p_m \\ & - \frac{p_m}{2u} \left((1 - p_m) \lg \frac{(1 - p_m)}{e} + (2u - (1 - p_m)) \right. \\ & \left. \lg \frac{2(2u - (1 - p_m))}{e} \right). \end{aligned} \quad (1.23)$$

Esta equação pode ser resolvida para o valor ótimo u_{ot} , o qual é dependente

¹³Estes valores são obtidos a partir de inúmeros testes, os quais não entraremos em detalhes.

p_m	u_{ot}	$\delta(u_{ot})$	$\delta(0.75)$	$\delta(0.5)$
0.02	0.6840	0.3662	0.3718	0.4361
0.04	0.6700	0.3018	0.3093	0.3589
0.06	0.6671	0.2529	0.2607	0.2999
0.08	0.6598	0.2144	0.2230	0.2533
0.10	0.6535	0.1836	0.1929	0.2162
0.12	0.6483	0.1592	0.1688	0.1868
0.14	0.6444	0.1400	0.1497	0.1639
0.16	0.6422	0.1253	0.1348	0.1466
0.18	0.6418	0.1145	0.1235	0.1340
0.20	0.6435	0.1072	0.1154	0.1258
0.22	0.6475	0.1029	0.1101	0.1213
0.24	0.6538	0.1013	0.1072	0.1190
0.26	0.6627	0.1019	0.1066	0.1220
0.28	0.6740	0.1045	0.1078	0.1265
0.30	0.6878	0.1087	0.1108	0.1334

Tabela 1.6: Valores para otimização do intervalo de normalização $[u, 2u)$.

de p_m . A tabela 1.6 mostra vários valores de u_{ot} em relação a valores de p_m . Ela também indica o valor correspondente de $\delta(u_{ot})$. Para facilitar a implementação o valor de u deve manter-se fixo, por exemplo, 0.75 ou 0.5. Pela tabela 1.6 notamos que para $u = 0.75$, $\delta(u)$ se aproxima muito mais de $\delta(u_{ot})$ do que para $u = 0.5$. Poderíamos, porém, escolher o intervalo $[0.625, 1.25)$, isto deixaria o processo um pouco mais complexo, para um ganho mínimo na compressão. Logo, recomenda-se o uso do intervalo $[0.75, 1.5)$ para tal. Os valores de u_{ot} para $p_m > 0.3$ não foram listados, pois para estes valores

Comprimento da entrada	Tamanho do alfabeto	Comprimento do código por símbolo	Entropia por símbolo	Eficiência
652	37	4.561	4.406	0.97
18996	84	4.842	4.726	0.98
71	4	1.718	1.669	0.97
71	4	1.394	1.386	0.99

Tabela 1.7: Eficiência para 4 entradas distintas com alfabetos distintos.

ocorre $(1 - p_m) < u_{ot}$ violando a condição para a derivação de 1.23. Mesmo assim, é possível derivar 1.23 para $u < (1 - p_m)$ e computar u_{ot} para estes valores. Felizmente para estes valores de p_m os valores de u_{ot} variam de 0.7472($p_m = 0.36$) à 0.70($p_m = 0.8$), logo, a escolha de $u = 0.75$ é certamente a melhor para a maioria dos casos.

Capítulo 2

Modelos Adaptativos

No capítulo anterior, vimos que a codificação aritmética requer, a cada passo, uma probabilidade para o símbolo a ser codificado. Surge então uma questão: *Como obter probabilidades para cada instância no processo ?*

Como resposta a esta questão surge o conceito de *modelo*, cujo papel principal é caracterizar uma fonte de informação, isto é, encontrar o mecanismo que esta fonte utiliza para gerar os símbolos. Seja, por exemplo, uma fonte que gera símbolos em ordem alfabética, ou seja, gera cadeias do tipo “a,b,c,d,e,...”. Um modelo bom para esta fonte é aquele que estima o próximo símbolo da ordem alfabética com alta probabilidade, e os demais símbolos com probabilidades próximas de zero. Podemos verificar, que qualquer outro modelo diferente deste não caracterizaria bem esta fonte.

Sendo assim, a função de um modelo é estimar probabilidades para cada símbolo, sendo que esta deve se aproximar o máximo da probabilidade real (dada pela fonte). A partir desta estimativa, a codificação aritmética, vista anteriormente, é utilizada para a codificação desta estimativa e obtenção de uma boa compressão da mensagem. Veremos que quanto melhor o modelo representa a fonte, maior compressão se obtém, o que torna o estudo de modelos fundamental no campo das compressões.

Inicialmente, iremos definir dois tipos de modelos, que são os mais utilizados no campo da codificação aritmética. Eles são :

- *Modelo de contexto-finito*
- *Modelo de estados finitos*

O *modelo de contexto-finito*¹ baseia-se em estimar a probabilidade do próximo símbolo a partir dos σ símbolos anteriores, sendo denominado por modelo de *ordem- σ* . Por exemplo, seja M a mensagem : “O livro $q\bullet$ ”, e tomemos $\sigma = 1$; o próximo símbolo $\bullet$ será estimado com a condição de que o último símbolo recebido foi q . Analisando a língua portuguesa, vemos que o símbolo u é o símbolo de maior probabilidade após q . Sendo assim, se o modelo estiver caracterizando uma fonte que gera palavras em português, ele deve estimar u com grande probabilidade e os outros símbolos com probabilidades menores, atingindo deste modo o seu objetivo que é obter uma estimativa para a probabilidade o mais próximo da probabilidade real dada pela fonte. Os modelos de contexto-finito serão descritos mais detalhadamente nas próximas seções, além de nossa implementação, que usa este tipo de modelo.

Os *modelos de estados finitos*, freqüentemente chamados de *modelos de Markov*, nome devido ao matemático A.A.Markov (1856-1922), são baseados em *máquinas de estados finitos*. Uma máquina de estados finitos é caracterizada por um conjunto de estados Q , um alfabeto A e uma função transição δ , que a partir de um símbolo a do alfabeto A , leva o estado Q_i a um novo estado Q_j , isto é, $\delta(Q_i, a) = Q_j$. Quando utilizamos a máquina de estados finitos para modelar uma fonte, definimos também a função probabilidade p relativa a uma transição δ , isto é, se estivermos em um estado Q_i a probabilidade de executarmos a transição $\delta(Q_i, a)$ vale $p(\delta(Q_i, a))$. O modelo deve possuir um estado inicial Q_0 e, a cada símbolo processado, a transição é executada e a estimativa de probabilidade é obtida a partir da probabilidade relativa a esta transição. Este tipo de modelagem pode ser aplicado, como exemplo, a cadeias de DNA. Uma cadeia de DNA é formada por bases (A,C,G,T), que se agrupam em triplas; onde a probabilidade de uma base ocorrer está intimamente ligada a posição que ela ocupa na tripla. A figura 2.1 exhibe uma visualização de um modelo de estados finitos para as cadeias de DNA. Em [4] temos uma implementação de um algoritmo que usa modelos de estados finitos para a representação da fonte, este algoritmo é conhecido por DMC (*Dynamic Markov Compression*).

Como já dissemos os modelos de contexto-finito também podem ser caracterizados usando o de estados finitos. Por exemplo, um modelo de contexto-

¹Veremos mais adiante que este modelo é um caso particular do modelo de estados finitos, porém ele é estudado separadamente, e os algoritmos que o utilizam são agrupados em uma classe diferente dos que utilizam o modelo de estados finitos.

Figura 2.1: Exemplo de um modelo de estados finitos para cadeia de DNA.

finito de *ordem-0* pode ser caracterizado por um modelo com um estado único e auto-transições para cada símbolo. Um modelo de *ordem-1* pode ser caracterizado por um modelo com m (número de símbolos no alfabeto) estados, cada um com m transições, onde cada estado representa o último símbolo executado. Um modelo de *ordem- σ* é caracterizado por m^σ estados, cada um com m transições, onde cada estado representa os σ últimos símbolos executados.

Feitas as definições acima, calcularemos a *entropia* de um modelo. O conceito de entropia será usado como a quantidade de informação média por símbolo gerado pela fonte². Pela maior facilidade de cálculo, faremos o cálculo em relação ao modelo de estados finitos. Definimos a entropia E_i de um estado Q_i como $-\sum_{k \in A} p(\delta(Q_i, k)) \lg p(\delta(Q_i, k))$, onde A é o alfabeto. Seja P_i a probabilidade de ocorrer o estado Q_i , logo, a entropia do modelo é calculada por $\sum_{i \in Q} P_i E_i$, onde Q é o conjunto de estados do modelo. Como exemplo, podemos tomar a figura 2.1. Se assumirmos que os estados são equiprováveis, então a entropia será :

²Expressaremos esta quantidade por *bits/símbolo*.

$$\begin{aligned}
& \frac{0.25 \lg 0.25 + 0.25 \lg 0.25 + 0.25 \lg 0.25 + 0.25 \lg 0.25}{3} - \\
& \frac{0.1 \lg 0.1 + 0.4 \lg 0.4 + 0.3 \lg 0.3 + 0.2 \lg 0.2}{3} - \\
& \frac{0.2 \lg 0.2 + 0.2 \lg 0.2 + 0.1 \lg 0.1 + 0.5 \lg 0.5}{3} = \\
& 1.8691 \text{ bits/simb.}
\end{aligned}$$

Conforme observamos no capítulo anterior, o nosso interesse é a obtenção da quantidade de informação de uma mensagem em particular, bastando para isto, somarmos a quantidade de informação relativa a cada transição a partir dos símbolos da mensagem, começando sempre pelo estado inicial Q_0 .

Definidos estes dois tipos de modelos, necessitamos, agora, estudar como é feita a construção de ambos. Esta construção, basicamente, pode ser feita de duas maneiras distintas, que são denominadas por :

- *Modelos não-adaptativos ou estáticos*
- *Modelos adaptativos ou dinâmicos*

Os *modelos não-adaptativos* são aqueles que caracterizam uma certa fonte fixa de informação; logo, quando não aplicado a esta fonte, pode não obter estimativas satisfatórias para as probabilidades. Além disto, é fundamental, tanto o codificador quanto o decodificador conhecerem o modelo que está sendo usado antes de executarem seus processos. Caso o decodificador não conheça o modelo que está sendo usado, o codificador deve colocar na saída, além do código, o modelo usado, e com isto o decodificador primeiro lê o modelo e depois executa o seu processamento. Os modelos não-adaptativos, também podem executar um pré-processamento, a fim de obter um modelo característico para a fonte, e uma vez construído, é necessário ser colocado na saída para o decodificador. Este tipo de construção é denominada por *modelo semi-adaptativo*. O nome semi-adaptativo se dá, pois este modelo constrói de uma maneira adaptativa um modelo não-adaptativo, isto é, estuda a mensagem a ser codificada e após este estudo tenta montar um melhor modelo para caracterizá-la.

Os *modelos adaptativos* são aqueles construídos iterativamente com os símbolos gerados pela fonte de informação, isto é, conforme mais símbolos são processados, um melhor modelo para representar a fonte pode ser obtido. Como o processo é iterativo, então, tanto o codificador como o decodificador podem executar a mesma tarefa, sendo desnecessária assim a transmissão do modelo usado.

O modelo não-adaptativo não é de grande uso em geral. No caso de não ser semi-adaptativo, só obtêm resultados satisfatórios para a fonte que ele caracteriza. Já no caso semi-adaptativo necessita-se de um pré-processamento para construir o melhor modelo característico para a fonte, e para uma transmissão contínua este pré-processamento fica limitado.

Os modelos adaptativos combinam o melhor das duas técnicas por executarem um simples passo seqüencial ao longo da mensagem, adaptando-o conforme a execução, ou seja, a cada passo uma instância é estimada de acordo com o modelo construído até o instante, e este atualizado a partir da *história*, que designa a parte da mensagem já processada. Este mecanismo é possível, pois, tanto o codificador como o decodificador têm acesso à mesma história e podem independentemente estimar a probabilidade do símbolo, e atualizar o modelo de maneira idêntica.

Veremos mais adiante que existem modelos não-adaptativos, na verdade semi-adaptativos, chamados de *modelos enumerativos* que obtêm, a partir de alguns parâmetros iniciais, estimativas exatas para cada símbolo da mensagem. Com isso, consegue-se melhores resultados do que os modelos adaptativos, pois, estes gastam tempo “aprendendo” o que este modelo já conhece da fonte. Em [2] é provado que os modelos adaptativos gastam no máximo mais S bits, onde S é o espaço necessário para se armazenar o modelo, de espaço para qualquer fonte do que o melhor modelo não-adaptativo. O modelo semi-adaptativo, por sua vez, constrói inicialmente o melhor modelo para a fonte, e depois o codifica. Esta codificação gasta espaço S bits de código, portanto, o código obtido pelo modelo adaptativo nunca têm maior comprimento que o obtido pelo modelo semi-adaptativo.

2.1 Classes de equivalência

A codificação de uma mensagem $M = x_1x_2..x_t$ leva em conta a sua probabilidade $p(M)$, que pode ser calculada, como foi visto anteriormente, de acordo

com um modelo. Determinada sua probabilidade, vimos no capítulo anterior que M possui uma quantidade de informação igual a $-\lg p(M)$, valor este que a codificação aritmética pode atingir para um código de M^3 . Infelizmente, calcular a probabilidade de uma mensagem inteira para posterior codificação é um procedimento impraticável. Logo, para efetivamente conseguirmos codificar a probabilidade de uma mensagem, devemos decompor o processo, e codificar probabilidades intermediárias da mensagem denominadas de *probabilidades condicionais*.

Como vimos, os modelos adaptativos tentam a cada passo estimar uma probabilidade para cada símbolo da mensagem. Esta estimativa sempre é feita de acordo com uma probabilidade condicional entre o símbolo a ser processado e a palavra já recebida, ou seja, se $M_i = x_1x_2\dots x_i$ é a palavra recebida até o instante i e x_{i+1} é o próximo símbolo, então, a probabilidade da instância $(x_{i+1}, i+1)$ é $p(x_{i+1}, i+1) = p(x_{i+1} | M_i)$, denominada probabilidade condicional de x_{i+1} dada a ocorrência de M_i .

Uma nova função de probabilidade $p'(y)$ será definida como a probabilidade de uma palavra y ser o prefixo de alguma mensagem gerada pela fonte. Ela pode ser calculada pela soma de todas as probabilidades das mensagens às quais y é prefixo. Se λ é a palavra vazia então $p'(\lambda) = 1$, pois λ é prefixo de todas as mensagens.

Sendo assim, podemos decompor a probabilidade de uma mensagem $M = x_1x_2\dots x_t$ por

$$p(x_1\dots x_t) = \frac{p'(x_1)}{p'(\lambda)} \frac{p'(x_1x_2)}{p'(x_1)} \frac{p'(x_1x_2x_3)}{p'(x_1x_2)} \dots \frac{p'(x_1\dots x_t)}{p'(x_1\dots x_{t-1})} \frac{p(x_1\dots x_t)}{p'(x_1\dots x_t)}. \quad (2.1)$$

Esta decomposição é válida, pois, do lado direito todos os termos são cancelados, restando apenas $p(x_1\dots x_t)$. A probabilidade condicional aparece nesta decomposição pelos termos do lado direito, logo, é definida por

$$p(x_{i+1} | x_1\dots x_i) = \frac{p'(x_1\dots x_{i+1})}{p'(x_1\dots x_i)}. \quad (2.2)$$

³Como vimos, a codificação aritmética com precisão infinita atinge para um código de M um comprimento aproximado de $-\lg p(M)$.

Com isto, a probabilidade de uma mensagem pode ser decomposta pelas probabilidades condicionais de cada um de seus símbolos multiplicada pelo último fator de 2.1 ($p(x_1..x_t)/p'(x_1..x_t)$), expressão que representa a probabilidade de M ser uma mensagem, dividida pela probabilidade de M ser o prefixo de alguma mensagem. Se assumirmos que podemos sempre determinar o final da mensagem, então este termo será 1 (se o último símbolo for x_t) ou 0 (caso não esteja finalizada em x_t). Como vimos no capítulo anterior, é necessário sabermos quando a mensagem é finalizada. A primeira maneira é manter um símbolo especial de fim de mensagem, bastando apenas verificar se x_t é este símbolo. Uma segunda maneira é tratar apenas mensagens de comprimento fixo, ou transmitir o seu comprimento a priori, no caso deste comprimento valer t , então, o último termo de 2.1 valerá 1. Uma terceira opção é feita para o caso em que a transmissão é contínua e todas as mensagens são infinitamente longas, logo, todas as probabilidades de mensagens tendem a 0.

A decomposição em 2.1 ainda não ajuda, pois, os cálculos das probabilidades condicionais são exaustivos⁴, devendo consultar toda a história (veja 2.2). Por esta razão, devemos estudar um meio de aproximar tais probabilidades. Uma maneira geral de aproximá-las é ignorar parte da história. Pode-se, por exemplo, aproximar a probabilidade condicional de x_{i+1} levando em conta apenas os dois últimos símbolos, ou seja, $p(x_{i+1} | x_1..x_i) \approx p(x_{i+1} | x_{i-1}x_i)$. Isto faz com que o cálculo da probabilidade condicional não necessite verificar toda a história, e sim apenas os dois últimos símbolos. Podemos definir, assim, uma nova função U , que quando aplicada a uma palavra, retorna o grupo ao qual esta faz parte. Denominaremos este grupo por *classe de equivalência* da palavra. Qualquer função pode ser aplicada a uma palavra para se obter sua classe de equivalência, porém, na prática, apenas algumas são usadas. Para o nosso estudo, como usaremos modelos de contexto-finito, estudaremos apenas as funções que retornam os últimos símbolos da palavra. No exemplo, $U(x_1..x_i) = x_{i-1}x_i$, e, se m é o número de símbolos do alfabeto, então, existem m^2 classes de equivalência para este exemplo. Logo, a probabilidade condicional pode ser aproximada por

$$p(x_{i+1} | x_1..x_i) \approx p(x_{i+1} | U(x_1..x_i)). \quad (2.3)$$

⁴Veremos mais adiante uma maneira de calcular probabilidades exatas, e como é exaustivo este cálculo.

Verificando os modelos já apresentados, conseguimos identificar as classes de equivalência correspondentes a cada um deles. Nos modelos de estados finitos cada classe de equivalência é representada por um estado Q_i , logo, a partir de uma palavra conseguimos determinar a qual classe de equivalência ela pertence, bastando para isto, apenas percorrer as transições relativas aos símbolos da palavra a partir do estado inicial Q_0 .

Para um modelo de contexto-finito, a classe de equivalência de uma palavra é representada pelos seus últimos σ símbolos, sendo assim denominado de modelo de contexto-finito de *ordem*- σ . No decorrer deste capítulo, analisaremos alguns métodos para calcular probabilidades condicionais a partir das classes de equivalência.

Finalizadas as definições iniciais de modelos e estimativas de probabilidades, resumiremos o processo de codificação de uma mensagem, a um processo incremental, ou seja, o modelo é inicializado com uma distribuição de probabilidades, e, à medida que a mensagem vai sendo processada (tanto codificada como decodificada), ela atualiza o modelo, o qual retorna uma nova distribuição com o uso de suas classes de equivalência. O codificador usará um processo de codificação (por exemplo: *codificação aritmética*) para codificar o símbolo atual de acordo com sua probabilidade condicional. Este código é transmitido através de um canal de comunicação⁵. O decodificador, por sua vez, recebe o código transmitido. A decodificação é feita utilizando-se uma distribuição de probabilidades calculada pelo modelo com um processo análogo ao usado pelo codificador. Sempre após a codificação (ou decodificação) de um símbolo, o modelo é atualizado. A figura 2.2 mostra este processo.

2.2 Modelos semi-adaptativos

Iniciaremos nesta seção o estudo de modelos semi-adaptativos, os quais executam duas etapas durante o processo: construção do modelo e codificação propriamente dita. Para começar, introduziremos algumas notações iniciais. Seja $Z = \{0, 1, \dots, K - 1\}$ o conjunto de K classes de equivalência, onde z indica uma destas classes. Seja A um alfabeto com m símbolos, e M uma

⁵Vamos supor que este canal de comunicação é livre de erros, pois, caso contrário, teríamos que fazer o uso de um mecanismo de correção de erros, matéria da qual não iremos tratar neste trabalho

Figura 2.2: Processo incremental de codificação de uma mensagem.

mensagem de comprimento t . Seja $c^z(M)$ o número de vezes em que a classe z é seguida por algum símbolo de A ao longo da mensagem M , e $c^{zx}(M)$ o número de vezes que a classe z é seguida por um símbolo $x \in A$ em M . Com isto, temos a seguinte igualdade:

$$c^z(M) = \sum_{x \in A} c^{zx}(M). \quad (2.4)$$

Como nenhuma transição ocorre após a última classe de equivalência da mensagem, $c^z(M)$ pode ser uma unidade menor do que o número de vezes que a classe z aparece em M . As funções c^z e c^{zx} podem ser estendidas para palavras⁶ sobre o alfabeto A , tendo definições análogas às vistas para mensagens. Denominemos por $d^{zx}(y)$ o número de vezes que restam em M para uma palavra com classe z ser seguida pelo símbolo x , dado que já ocorreu a palavra y , ou seja, $d^{zx}(y) = c^{zx}(M) - c^{zx}(y)$. Do mesmo modo temos que $d^z(y)$ é o número de vezes que restam para a classe z ser seguida por algum símbolo de A ao longo da mensagem M , ou seja, $d^z(y) = c^z(M) - c^z(y)$.

Feitas as definições iniciais, podemos agora verificar todos os parâmetros necessários para a construção do modelo:

- O alfabeto A ;
- Um conjunto Z de classes de equivalência;
- Uma função U que leva palavras sobre A em classes de Z ;
- Um conjunto de frequências c^{zx} , para todos $z \in Z$ e $x \in A$.

Para nossa próxima análise assumiremos que a fonte gera apenas mensagens que satisfazem os parâmetros acima, e, além disto, elas são equiprováveis. Estas premissas fazem parte de um tipo de codificação chamada de *codificação enumerativa*. Para os próximos cálculos, devemos ainda observar algumas definições. Denominemos por $N(y)$ o número de vezes que uma palavra y aparece como prefixo de alguma mensagem possível, sendo $N(\lambda)$ igual ao número total de mensagens possíveis. Logo, a probabilidade

⁶Podemos entender que palavra é um prefixo de uma mensagem, ou seja, um pedaço não finalizado da mensagem gerada pela fonte.

de y ser um prefixo de alguma mensagem é $p'(y) = N(y)/N(\lambda)$. Usando a equação 2.2 temos que as probabilidades condicionais podem ser escritas por:

$$p(x_{i+1} \mid x_1..x_i) = \frac{N(x_1..x_{i+1})}{N(x_1..x_i)}. \quad (2.5)$$

A seguir veremos os vários métodos de se obter probabilidades para este tipo de modelo.

2.2.1 Método exato

Uma primeira maneira de se obter probabilidades para modelos semi-adaptativos é o método exato. Este se caracteriza por calcular probabilidades exatas a partir dos parâmetros vistos anteriormente. Daremos dois exemplos de como se calcular estas probabilidades. O primeiro será feito sobre um modelo de *ordem-0*, onde somente existe uma classe de equivalência.

Tomemos o conjunto completo de mensagens do alfabeto $0, 1$, onde cada mensagem possui um número c^0 de “0”s e um número c^1 de “1”s. O comprimento de cada mensagem é $t = c^0 + c^1$. O número $N(y)$ de vezes que uma palavra y é prefixo de alguma mensagem equivale à expressão:

$$N(y) = \frac{(t - |y|)!}{d^0(y)!d^1(y)!}, \quad (2.6)$$

onde $|y|$ é o comprimento da palavra y . Usando 2.5 temos que a probabilidade de um símbolo b ocorrer, dado que ocorreu y , é:

$$p(b \mid y) = \frac{d^b(y)}{t - |y|} \quad (2.7)$$

que equivale ao número de símbolos b que faltam ocorrer ($d^b(y)$) dividido pelo número total de símbolos que faltam para terminar a mensagem ($t - |y|$).

Vejamos agora um modelo de *ordem-1*, onde existe uma classe de equivalência para cada símbolo do alfabeto A . Tomemos o conjunto completo de mensagens do alfabeto $\{0, 1\}$ que possuem $c^{b_1 b_2}$ subsequências $b_1 b_2$, onde

$b_1, b_2 \in \{0, 1\}$. O comprimento de cada mensagem é $t = c^{00} + c^{01} + c^{10} + c^{11}$ (devemos considerar a classe inicial). Para determinarmos o valor de $N(y)$, devemos verificar o último símbolo w de y e calcular o número de combinações possíveis dos símbolos restantes restringidos ao número $d^{b_1 b_2}$ de subsequências restantes. Calculando estas combinações obtemos:

$$N(y'w) = \frac{(d^{ww} + d^{\bar{w}w})! (d^{\bar{w}\bar{w}} + d^{w\bar{w}} - 1)!}{d^{ww}! d^{\bar{w}w}! d^{\bar{w}\bar{w}}! (d^{w\bar{w}} - 1)!} \quad (2.8)$$

onde $\bar{w}$ é o símbolo oposto de w , e y' é a palavra obtida de y retirando-se o último símbolo w .

Verificamos a partir de 2.5 que a probabilidade condicional $p(b \mid y'w)$ depende somente de b e w . Logo, existem três casos : quando $b = w$, quando $b = \bar{w}$, e quando $y = \lambda$ (início da mensagem). Logo, temos:

$$\begin{aligned} p(w \mid y'w) &= \frac{d^{ww}}{d^{ww} + d^{\bar{w}w}}, \\ p(\bar{w} \mid y'w) &= \begin{cases} \frac{d^{\bar{w}w}}{d^{ww} + d^{\bar{w}w}}, & d^{ww} > 0 \\ 1, & d^{ww} = 0, \end{cases} \\ p(w \mid \lambda) &= \frac{d^{w\bar{w}}(d^{ww} + d^{\bar{w}w})}{d^{w\bar{w}}(d^{ww} + d^{\bar{w}w}) + d^{\bar{w}\bar{w}}(d^{\bar{w}\bar{w}} + d^{w\bar{w}})}. \end{aligned} \quad (2.9)$$

Observa-se claramente que para modelos de *ordem-1* a complexidade do cálculo das probabilidades cresceu consideravelmente, logo, para modelos de ordens maiores este cálculo pode se tornar uma tarefa muito complexa. Por esta razão, aproximações no cálculo são necessárias, sendo que estas devem ser feitas, de maneira a obter probabilidades satisfatórias tanto do ponto de vista de eficiência nas estimativas, quanto de complexidade no cálculo.

2.2.2 Método aproximado

O método aproximado é uma segunda maneira de se calcular probabilidades condicionais. Dada uma palavra y e uma classe de equivalência z , conseguimos obter, a partir de uma inspeção, a ordem de ocorrência de cada um

dos $c^z(y)$ símbolos que seguem a classe z em y . Inversamente, dadas as ordenações dos símbolos que seguem cada classe, obtemos uma única mensagem correspondente. Para isto, tome a classe inicial $z = U(\lambda)$ e seja a_1 o primeiro símbolo na ordenação da classe z . Inicie a mensagem com a_1 . Execute o mesmo processo com o novo $z = U(a_1)$, tomando como símbolo seguinte da mensagem o próximo símbolo da ordenação da classe z . Repita, iterativamente, o procedimento até finalizar todos os símbolos de cada ordenação.

O método aproximado se baseia em verificar a quantidade total de ordenações possíveis dos símbolos para cada classe de equivalência, com base nos parâmetros iniciais, sendo que o produto entre o número de ordenações para cada classe, indica uma quantidade “aproximada” de mensagens possíveis. Este cálculo é aproximado, pois, existem conjuntos de ordenações correspondentes a mensagens não válidas. Por exemplo, considere um modelo de *ordem-1* sobre o alfabeto binário $A = \{0, 1\}$, com os parâmetros $c^{00} = c^{01} = c^{10} = 2$, $c^{11} = 1$, duas classes de equivalência $\{0, 1\}$, e $U(\lambda) = 0$. Seja a mensagem $M = 0011010$, a qual satisfaz o modelo e possui ordenação $(0, 0, 1, 1)$ para a classe 0 e $(1, 0, 0)$ para a classe 1. Considerando a ordenação $(1, 1, 0, 0)$ para a classe 0 e $(0, 0, 1)$ para a classe 1, obtemos a palavra $y = 101000$, a qual não pode ser completada, já que falta uma transição a ser processada na classe 1, $U(y) = 0$ e todas as transições da classe 0 já foram executadas. Sendo assim, estas ordenações correspondem a uma mensagem não válida.

A quantidade de ordenações possíveis para uma classe específica z é calculada pela permutação dos c^z símbolos da ordenação, dividida pelo produto das permutações dos c^{zx} símbolos x . Isto é:

$$\frac{c^z!}{\prod_{x \in A} c^{zx}!}. \quad (2.10)$$

A função de enumeração aproximada $N^\sim(y)$ para o método aproximado é obtida através do produto do número de ordenações possíveis dos $d^z(y)$ símbolos restantes em cada classe. Logo, temos:

$$N^\sim(y) = \prod_{z \in Z} \frac{d^z(y)!}{\prod_{x \in A} d^{zx}(y)!}. \quad (2.11)$$

E a partir de 2.5 encontramos a probabilidade condicional correspondente por:

$$p^{\sim}(b \mid y) = \frac{d^{zb}(y)}{d^z(y)}, \quad (2.12)$$

onde $z = U(y)$.

2.2.3 Método fixo

Este último método é o mais simples de todos, sendo usado apenas em aplicações práticas de codificação, as quais necessitam de pouco processamento e muita rapidez. Ele se baseia em assumir que a frequência de cada símbolo é constante, logo, a probabilidade condicional de cada símbolo x dada uma classe z , é calculada pelo número de ocorrências c^{zx} do símbolo nesta classe, dividido pelo número de ocorrências c^z da própria classe, ou seja:

$$p^F(b \mid y) = \frac{c^{zb}}{c^z}, \quad (2.13)$$

onde $z = U(y)$. De 2.5 pode-se obter a função de enumeração N^F para o método fixo, que vale:

$$N^F(y) = \prod_{z \in Z} \frac{(c^z)^{d^z(y)}}{\prod_{x \in A} (c^{zx})^{d^{zx}(y)}}. \quad (2.14)$$

A tabela 2.1 mostra um exemplo que inclui os métodos: exato, aproximado e fixo. Ela usa um modelo de *ordem-1* sobre o alfabeto binário $A = \{0, 1\}$, sendo $Z = \{0, 1\}$, e assume que a classe $U(\lambda) = 0$. A tabela mostra as funções $d^z(y)$ e $d^{zx}(y)$ para cada prefixo y já processado da mensagem, e sua respectiva probabilidade condicional. Por este exemplo, observamos que o método exato obtém melhor resultado que o aproximado, que por sua vez, melhor que o fixo.

	y_1	y_2	y_3	y_4	y_5	y_6	y_7	
	0	0	1	1	0	1	0	
$d^0(y)$	4	3	2	1	1	1	0	
$d^1(y)$	3	3	3	3	2	1	1	
$d^{00}(y)$	2	1	0	0	0	0	0	
$d^{01}(y)$	2	2	2	1	1	1	0	
$d^{10}(y)$	2	2	2	2	2	1	1	
$d^{11}(y)$	1	1	1	1	0	0	0	
Métodos								$P(M = y)$ Bits
Exato [eq. 2.9]	$\frac{6}{12}$	$\frac{1}{3}$	1	$\frac{1}{2}$	1	1	1	$\frac{1}{12}$ 3.58
Aproximado [eq. 2.12]	$\frac{2}{4}$	$\frac{1}{3}$	1	$\frac{1}{3}$	1	1	1	$\frac{1}{18}$ 4.17
Fixo [eq. 2.13]	$\frac{2}{4}$	$\frac{2}{4}$	$\frac{2}{4}$	$\frac{1}{3}$	$\frac{2}{3}$	$\frac{2}{4}$	$\frac{2}{3}$	$\frac{1}{108}$ 6.75

Tabela 2.1: Exemplo dos métodos exato, aproximado e fixo para um modelo de ordem-1.

2.2.4 Análise comparativa

Faremos, agora, uma breve análise entre os métodos já vistos. Inicialmente, vamos calcular o tamanho S necessário para representar um modelo. Como vimos, um modelo é composto por K classes de equivalência, sendo que cada classe possui m contadores c^{zx} (um para cada um dos m símbolos de A). Cada um desses contadores deve conseguir armazenar um número de no máximo o comprimento total t da mensagem, ou seja, $O(\log t)$. Logo, temos:

$$S = O(Km \log t). \quad (2.15)$$

Com esta definição, mostraremos que o limite da diferença entre o comprimento do código obtido pelo método exato e os outros dois (aproximado e fixo), a medida que o comprimento t da mensagem cresce, é $O(Km \log t)$.

Para mostrar que um código C obtido pelo método exato é menor que um código $C^\sim$ obtido pelo método aproximado, basta observarmos que $N(\lambda) \leq N^\sim(\lambda)$. Isto porque ambas funções, $N(\lambda)$ e $N^\sim(\lambda)$, indicam a quantidade total de mensagens para cada método, e como cada mensagem é equiprovável, vale que $C = \lg N(\lambda)$ bits e $C^\sim = \lg N^\sim(\lambda)$ bits. Para mostrar que $N(\lambda) \leq$

$N^\sim(\lambda)$ basta observar que este último, como já foi visto, enumera um conjunto maior que o total de mensagens válidas, e o primeiro enumera somente mensagens válidas. Da mesma maneira, têm-se que o código $C^\sim$ é menor que o código C^F , obtido pelo modelo fixo, já que para cada termo entre 2.11 e 2.14 vale a seguinte desigualdade:

$$\frac{c^z!}{\prod_{x \in A} c^{zx}!} \leq \frac{(c^z)^{c^z}}{\prod_{x \in A} (c^{zx})^{c^{zx}}}. \quad (2.16)$$

Ela pode ser demonstrada, facilmente, pela decomposição do primeiro termo. Assim, vale que $N^\sim(\lambda) \leq N^F(\lambda)$.

A eficiência relativa entre o método fixo e o aproximado pode ser comparada encontrando-se um limite superior para a diferença entre C^F e $C^\sim$. Com o uso da aproximação de Stirling em 2.11 e 2.14, temos que:

$$C^F - C^\sim \leq \frac{1}{2} \left[K(m-1) \left(\lg 2\pi + \frac{1}{6} \right) + \sum_{z \in Z, x \in A} \lg c^{zx} - \sum_{z \in Z} \lg c^z \right]. \quad (2.17)$$

A diferença entre as duas últimas somatórias terá seu valor máximo quando $c^z = t/K$ e $c^{zx} = t/Km$, isto implica em:

$$C^F - C^\sim \leq \frac{1}{2} \left[K(m-1) \left(\lg 2\pi + \frac{1}{6} \right) + K(m-1)(\lg t - \lg K) + Km \lg m \right]. \quad (2.18)$$

Assumindo que o comprimento da mensagem é bem maior que o número de classes de equivalências, isto é, $t \gg K$, obtemos o seguinte limite superior:

$$C^F - C^\sim \leq \frac{1}{2} K(m-1) \lg t + O(Km \lg(m/K)). \quad (2.19)$$

Devemos agora calcular um limite superior para a diferença entre o código obtido pelo método exato e pelo aproximado ($C^\sim - C$). A dificuldade neste

caso, porém, está em calcular um limite inferior para $N(\lambda)$. É provado em [2], que a partir de algumas suposições conseguimos obter a seguinte inequação:

$$p(b \mid y) \leq \frac{d^{zb}}{d^z - 1}, \quad (2.20)$$

onde $z = U(y)$, e, a partir da equação 2.5, isto implica em:

$$N(\lambda) \geq \prod_{z \in Z} \frac{(c^z - 1)!}{\prod_{x \in A} c^{zx}!}, \quad (2.21)$$

que combinada com 2.11 e re-fomulada, resulta em:

$$\lg N^{\sim}(\lambda) - \lg N(\lambda) \leq \sum_{z \in Z} \lg c^z. \quad (2.22)$$

Assumindo que $t \gg K$, o máximo para este limite é atingido quando todos os c^z valem t/K . Logo, o limite superior para a diferença entre os códigos aproximado e exato vale:

$$C^{\sim} - C \leq K \lg t - K \lg K. \quad (2.23)$$

Combinando as equações 2.19 e 2.23 obtemos o limite para a diferença entre o método fixo e o exato. A tabela 2.2 resume todos os resultados em sua forma assintótica.

Os C , $C^{\sim}$ e C^F , obtidos até agora, não incluem o tamanho S necessário para transmitir o modelo, ou seja, os contadores c^{zx} . Devemos salientar contudo, que para todos estes modelos, a inclusão destes parâmetros é necessária, fazendo assim com que cada um dos códigos seja acrescido de S bits. Denominaremos estes códigos finais por E , $E^{\sim}$ e E^F , onde:

$$E = C + S,$$

$$\begin{array}{c}
C \leq C^\sim \leq C^F \\
C^\sim - C \leq K \lg t \\
C^F - C \leq \frac{K(m+1)}{2} \lg t \\
C^F - C^\sim \leq \frac{K(m-1)}{2} \lg t
\end{array}$$

Tabela 2.2: Comparação entre os códigos obtidos pelos métodos exato, aproximado e fixo em sua forma assintótica.

$$\begin{aligned}
E^\sim &= C^\sim + S, \\
E^F &= C^F + S.
\end{aligned}$$

2.3 Modelos adaptativos

Discutiremos, nesta seção, um tipo de modelagem um pouco distinta da anterior. A diferença agora, é que nem o codificador nem o decodificador, conhecem o conjunto de frequências c^{zx} das mensagens a serem codificadas, sendo que, os únicos parâmetros iniciais para ambos são: o alfabeto A com seus m símbolos, uma função U que dá a classe de equivalência de uma palavra sobre A , um conjunto Z das classes de equivalência associadas a U , e, possivelmente, um limite superior t_{max} para o comprimento das mensagens a serem codificadas. Poderíamos, neste caso, enumerar todas as possíveis mensagens, porém não obteríamos compressão alguma já que cada mensagem sendo equiprovável, o código resultante teria comprimento da ordem de t_{max} . Como as mensagens válidas possuem geralmente comprimentos menores que t_{max} , uma enumeração causará, neste caso, a expansão ao invés de compressão.

A principal idéia do código adaptativo não é transmitir os contadores c^{zx} , e sim acumulá-los ao longo do processo. Sendo assim, os contadores acumulados se adaptam aos contadores reais obtidos pela codificação semi-adaptativa. Logo, se uma palavra y já foi codificada, então tanto o codificador como o decodificador conhecem $c^{zx}(y)$, para $z \in Z$ e $x \in A$, porém, não conhecem $d^{zx}(y)$. Por analogia com 2.5, podemos estimar as probabilidades

adaptativas por:

$$p^{\text{adap}}(w|y) = \frac{c^{zw}(y)}{c^z(y)}, \quad (2.24)$$

onde $z = U(y)$ e $w \in A$, e agora, os contadores são obtidos da parte da mensagem já processada, ao invés do que ainda será processado.

Este cálculo de probabilidade, porém, é insuficiente, já que $c^{zw}(y)$ pode valer zero, e assim a probabilidade condicional correspondente será nula, e se por ventura um símbolo com esta probabilidade ocorrer, ele não poderá ser codificado. Este problema é denominado *problema da frequência zero*, e no campo dos códigos adaptativos, é a questão que mais deve ser analisada. Para resolvê-lo, devemos reservar espaço de probabilidade para todos os símbolos que podem, mas não ocorreram ainda dentro de cada classe z . Estudaremos, nesta seção, dois métodos para códigos adaptativos que resolvem o problema da frequência zero e são de fácil análise. Para facilitar a análise dos próximos códigos, denominaremos, simplesmente, por c^{zx} e c^z o valor destes contadores sobre a parte y já processada da mensagem (omitindo freqüentemente o parâmetro y). Introduziremos, também, um novo contador q^z que informa o número de símbolos distintos que seguem a classe z ao longo da parte da mensagem já processada, isto é:

$$q^z = |x \in A : c^{zx} > 0|, \quad (2.25)$$

onde $z \in Z$.

2.3.1 Método adaptativo A

Este primeiro método é caracterizado por reservar um espaço unitário para todos os símbolos que ainda não ocorreram em cada classe z . Logo, a probabilidade de um símbolo w que já ocorreu em z vale:

$$\frac{c^{zw}}{c^z + 1} \text{ ao invés de } \frac{c^{zw}}{c^z}. \quad (2.26)$$

Como o tamanho do alfabeto é m , então, seguindo a classe z ainda não ocorreram $m - q^z$ símbolos em y , logo, a probabilidade de um símbolo w que ainda não ocorreu seguindo z em y é

$$\frac{\frac{1}{c^z+1}}{m - q^z}. \quad (2.27)$$

Sendo assim, temos a probabilidade condicional:

$$p^A(w|y) = \begin{cases} \frac{c^{zw}(y)}{c^z(y)+1}, & c^{zw}(y) > 0 \\ \frac{1}{(c^z(y)+1)(m-q^z(y))}, & c^{zw}(y) = 0. \end{cases} \quad (2.28)$$

onde $z = U(y)$.

2.3.2 Método adaptativo B

Neste método, é reservado, inicialmente, para cada símbolo um espaço unitário dentro de cada classe de equivalência. Deste modo, os contadores para cada símbolo w em z valem $(c^{zw} + 1)$, totalizando um valor de $(c^z + m)$ para cada classe z . Temos, assim, a seguinte probabilidade condicional:

$$p^B(w|y) = \frac{c^{zw}(y) + 1}{c^z(y) + m}, \quad (2.29)$$

onde $z = U(y)$. A tabela 2.3 mostra o mesmo exemplo da tabela 2.1, para os métodos adaptativos A e B, sobre um alfabeto binário $A = \{0, 1\}$ e fazendo uso de um modelo de ordem-1. Neste exemplo, também assumimos $U(\lambda) = 0$.

2.3.3 Análise comparativa

Faremos agora uma análise entre os métodos adaptativos A e B, em comparação aos métodos semi-adaptativos, a fim de obter um limite superior e inferior para ambos.

	y_1	y_2	y_3	y_4	y_5	y_6	y_7	
	0	0	1	1	0	1	0	
$q^0(y)$	0	1	1	2	2	2	2	
$q^1(y)$	0	0	0	0	1	2	2	
$c^0(y)$	0	1	2	3	3	3	4	
$c^1(y)$	0	0	0	0	1	2	2	
$c^{00}(y)$	0	1	2	2	2	2	2	
$c^{01}(y)$	0	0	0	1	1	1	2	
$c^{10}(y)$	0	0	0	0	0	1	1	
$c^{11}(y)$	0	0	0	0	1	1	1	
Métodos								$P(M = y)$ Bits
A [eq. 2.28]	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{3}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{3}$	$\frac{1}{576}$ 9.17
B [eq. 2.29]	$\frac{1}{2}$	$\frac{2}{3}$	$\frac{1}{4}$	$\frac{1}{2}$	$\frac{1}{3}$	$\frac{2}{5}$	$\frac{2}{4}$	$\frac{1}{360}$ 8.49

Tabela 2.3: Exemplo dos métodos adaptativos A e B para um modelo de ordem-1.

No método A, quando o símbolo w vai ser codificado, ele contribui com $-\lg p^A(w|y)$ que vale $\lg((1 + c^z)/c^{zw})$ se w já tiver ocorrido na classe z , ou $\lg((1 + c^z)(m - q^z))$ se é a primeira ocorrência de w em z [equação 2.28]. Ao analisarmos a contribuição de cada símbolo de uma mensagem M no comprimento E^A do código obtido pelo seu processamento, vemos que para cada classe z , cada um dos valores $1, 2, \dots, c^z$ aparece uma vez como divisor, e para cada símbolo w que segue z , os valores $1, 2, \dots, (c^{zw} - 1)$ aparecem uma vez como dividendo e os valores $m - q^z, \dots, m$ uma vez como divisor. Logo, o comprimento E^A do código obtido pelo método adaptativo A vale:

$$E^A = \lg \prod_{z \in Z} \frac{m!}{(m - q^z)!} \frac{c^z!}{\prod_{w \in A} (c^{zw} - 1)!}. \quad (2.30)$$

O último termo é aproximadamente igual ao número $N^{\sim}(\lambda)$ de mensagens possíveis obtido pelo método aproximado [equação 2.11], ou seja, ele é igual a $C^{\sim} + \sum_{z \in Z; w \in A} \lg c^{zw}$. De outro lado, podemos aproximar $m!/(m - q^z)!$ por m^{q^z} . Logo, ordenando os termos, obtemos:

$$E^A \leq C^\sim + \sum_{z \in Z; w \in A} \lg c^{zw} + \lg m \sum_{z \in Z} q^z. \quad (2.31)$$

Podemos verificar que as duas últimas somas atingem valor máximo quando cada um dos c^{zw} valem t/Km , com isto, temos:

$$\begin{aligned} E^A &\leq C^\sim + Km \lg t - Km \lg Km + Km \lg m \\ &\leq C^\sim + Km(\lg t - \lg K), \end{aligned} \quad (2.32)$$

sendo t o comprimento da mensagem, K o número de classes de equivalência e m o tamanho do alfabeto. Para obtermos um limite inferior, fazemos a mesma derivação da equação 2.30 porém sem aproximar $m!/(m - q^z)!$:

$$E^A = C^\sim + \sum_{z \in Z; w \in A} \lg c^{zw} + \sum_{z \in Z} \lg \frac{m!}{(m - q^z)!}. \quad (2.33)$$

As duas últimas somas atingem mínimo quando algum c^{zw} vale t e o resto vale 0, implicando em:

$$E^A \geq C^\sim + \lg t + \lg m. \quad (2.34)$$

O método B pode ser analisado do mesmo modo usado para o método A. Ao analisarmos a contribuição de cada símbolo de uma mensagem M no comprimento E^B do código obtido pelo seu processamento, observamos que para uma classe particular z , cada um dos valores $m, m + 1, \dots, m + c^z - 1$ aparece uma vez como divisor, e, para cada símbolo w , que ocorre em uma classe z , cada um dos valores $1, 2, \dots, c^{zw}$ aparece como dividendo. Somando-se para todas as classes z , temos:

$$E^B = \lg \prod_{z \in Z} \frac{(m + c^z - 1)!}{(m - 1)! \prod_{w \in A} c^{zw}!}. \quad (2.35)$$

Usando a equação 2.11, da mesma maneira que no método A, ficamos com:

$$E^B = C^\sim + \lg \prod_{z \in Z} \frac{(m + c^z - 1)!}{(m - 1)!c^z!}. \quad (2.36)$$

A segunda soma atinge valor máximo quando todos c^z valem t/K , e valor mínimo quando algum c^z vale t e o restante vale 0. Sendo assim, obtemos as seguintes desigualdades:

$$E^B \geq C^\sim + \lg \frac{(t + m - 1)!}{(m - 1)!t!}, \quad (2.37)$$

$$E^B \leq C^\sim + K \lg \frac{(t/k + m - 1)!}{(m - 1)!(t/k)!}. \quad (2.38)$$

Quando $t \gg m$, podemos aproximar cada um dos logaritmos por $(m - 1) \lg t$, ficando com:

$$C^\sim + K(m - 1) \lg t \geq E^B \geq C^\sim + (m - 1) \lg t. \quad (2.39)$$

Uma vantagem dos métodos adaptativos é que seu pior caso não é muito “ruim”, isto é, o comprimento do código obtido nunca é muito maior que o da mensagem. Já os métodos não-adaptativos⁷ podem, no pior caso, obter um código de comprimento arbitrariamente maior que o da mensagem original. Para mostrar tal caso, basta escolhermos uma mensagem, que possui como instâncias aquelas com a menor probabilidade condicional relativa a classe de equivalência z correspondente, isto é, se a classe de equivalência inicial é z e w é o símbolo que possui menor probabilidade $p(w|z)$ então w é o primeiro símbolo da mensagem. Como $p(w|z)$ é mínimo, então, ele é menor ou igual a $1/m$. Logo, qualquer mensagem construída desta maneira, terá

⁷Serão vistos como os modelos semi-adaptativos estudados, com a ressalva de já possuir o modelo definido, e não construir através de um passo de processamento.

um código, construído pelo mesmo modelo, de comprimento no mínimo igual ao original. Como geralmente os modelos não-adaptativos comprimem certas mensagens muito eficientemente, então, para cada classe z existe um símbolo com probabilidade menor que $1/m$ (se não existir, todos os símbolos de todas as classes possuem probabilidade $1/m$ fazendo com que este modelo nunca comprima). Sendo assim, dependendo do valor desta probabilidade, o código obtido pode possuir comprimento arbitrariamente maior que a mensagem original.

Para os modelos adaptativos, por sua vez, é possível o cálculo de um limite superior para o comprimento do código obtido. Para o método A quando $t > m$, a equação 2.30 possui pior caso quando existe uma classe z com $c^z = t$ e $c^{zw} = t/m$, logo, E^A é limitado por:

$$E^A \leq \lg t! - m \lg \left(\frac{t}{m} - 1 \right)! + \lg m!. \quad (2.40)$$

No caso de mensagens muito grandes, onde $t \gg m$, este limite se aproxima de:

$$E^A \leq t \lg m + \frac{m+1}{2} \lg t. \quad (2.41)$$

Como o comprimento da mensagem original usa $\lg m$ bits por símbolo, então, o comprimento do código obtido pelo método A é no máximo o da mensagem original acrescido de um fator proporcional a $\lg t$.

Para o método B, o pior caso na equação 2.35 ocorre quando todas as classes de equivalência possuem $c^z = t/K$ e todos os contadores c^{zw} valem t/Km , implicando na seguinte inequação:

$$E^B \leq K \lg \left(\frac{t}{K} + m - 1 \right)! - Km \lg \frac{t}{Km}! - K \lg(m-1)!. \quad (2.42)$$

No caso de mensagens muito grandes, onde $t \gg m$, este limite se aproxima de:

$$E^B \leq t \lg m + \frac{K(m-1)}{2} \lg t. \quad (2.43)$$

Pelo mesmo modo, o comprimento do código obtido pelo método B é no máximo igual ao original acrescido de um fator proporcional a $\lg t$.

2.4 Modelos adaptativos de contexto-finito

Nesta seção, estudaremos uma maneira adaptativa de construir modelos de contexto-finito. Como já foi visto, um modelo de contexto-finito de *ordem- σ* é determinado por uma função de equivalência U , que quando aplicada a uma palavra y , retorna a classe de equivalência representada pelos últimos σ símbolos de y . Na seção anterior, vimos que as probabilidades condicionais, relacionadas a uma certa classe z em um modelo adaptativo, são obtidas pelos contadores c^{zx} de cada símbolo x já processados de y .

Intuitivamente podemos supor que para grandes mensagens, quanto maior for a ordem usada, melhores resultados são obtidos. Esta suposição é incorreta ao observarmos o problema da frequência zero, já que o mesmo surge como um obstáculo ao uso de grandes contextos no processo adaptativo. Isto se deve ao fato que, ao usarmos uma ordem elevada existirão inúmeras classes com contadores nulos, e neste caso, a estimativa não levará em conta a estrutura da mensagem. Por exemplo, se o modelo usado tiver ordem-8 e o alfabeto tiver m símbolos, então, o modelo possuirá m^8 classes, e assim, poderão ser necessárias m^9 estimativas usando algum método de frequência zero, isto é, em cada classe no máximo m símbolos serão assim estimados. Por outro lado, se utilizarmos contextos pequenos, a compressão não será muito grande, já que a estrutura da mensagem já processada será pouco analisada na estimativa dos símbolos restantes.

Uma maneira de resolver este dilema é aplicar uma “mistura” de ordens ao modelo. Para isto, fixamos uma ordem máxima σ e os símbolos são estimados através de uma combinação entre as estimativas para contextos menores ou iguais a σ . Ao longo desta seção veremos alguns métodos de se combinar estas estimativas, além de nossa própria implementação.

2.4.1 Mistura de estimativas

Para este novo estudo, serão introduzidas novas funções de equivalência U_i , as quais levam uma palavra y na sua classe de equivalência z_i , conforme uma ordem- i . Para modelos de contexto-finito, a função U_i leva y na classe z_i representada pelos últimos i símbolos de y . Definiremos, também, uma função especial U_{-1} , que levará uma palavra y na classe z_{-1} , na qual todos os símbolos são equi-prováveis. Se assumirmos que o alfabeto A possui m símbolos e $w \in A$, então, $p(w|z_{-1}) = 1/m$. O número total de classes de equivalência deste novo modelo é igual a soma do número de classes para cada ordem (de 0 à máxima σ) mais a classe z_{-1} , logo, vale $1 + m + m^2 + \dots + m^\sigma$.

Feitas estas definições iniciais, podemos então verificar como é feita a estimativa de um símbolo w para um modelo de ordem máxima σ . A probabilidade condicional de um símbolo w , dado que a palavra y ocorreu, equivale à soma ponderada das probabilidades condicionais para cada uma das ordens menores ou iguais a σ . Isto é:

$$p^{\text{mista}}(w|y) = \sum_{o=-1}^{\sigma} u_o p(w|z_o), \quad (2.44)$$

onde u_o é um peso atribuído a ordem o e $z_o = U_o(y)$. De agora em diante denotaremos a probabilidade condicional mista de um símbolo w , apenas por $p(w)$, e a probabilidade condicional de ordem o por $p_o(w)$, já estando implícitas a classe z_o e a palavra já processada y . Denominaremos os contadores $c^{z_o w}$ apenas por $c_o(w)$, e o c^{z_o} por C_o , logo:

$$C_o = \sum_{w \in A} c_o(w). \quad (2.45)$$

Como um exemplo inicial, calcularemos a probabilidade condicional $p_o(w)$ de um símbolo w por:

$$p_o(w) = \begin{cases} \frac{c_o(w)}{C_o}, & o = 0, \dots, \sigma, \\ \frac{1}{m}, & o = -1. \end{cases} \quad (2.46)$$

Com esta nova estimativa de probabilidades, vemos que o problema da frequência zero não existe mais, já que todos os símbolos possuem pelo menos uma probabilidade p_o não-nula em alguma classe entre as ordens -1 e σ , e como a sua probabilidade final é a soma ponderada de suas probabilidades condicionais p_o , então, para uma escolha adequada dos pesos u_o , ela é não-nula.

O único problema, agora, é a definição dos pesos u_o . Eles devem ser definidos de forma a nunca obter uma probabilidade final nula para algum símbolo w . Uma maneira de definí-los é assumir um conjunto de pesos fixos, e outra, é adaptá-los conforme o processo, dando maior peso a ordens maiores no decorrer do processo. Nenhuma destas escolhas, porém, levam em conta o fato que a importância de uma ordem o varia de acordo com sua classe z_o e suas frequências c_o , isto é, podem existir classes de ordens elevadas que ocorreram poucas vezes, as quais devem receber pesos pequenos; do mesmo modo classes de ordens menores que ocorreram muitas vezes, as quais devem receber pesos maiores no decorrer do processo. Um modo mais coerente, é alocar em cada classe z_o um espaço de código que indica a possibilidade do próximo símbolo w poder ser estimado por uma classe de ordem menor. Alocado este espaço, mostraremos que o peso relacionado a esta classe indicará a probabilidade do próximo símbolo ser estimado usando apenas estimativas da mesma.

O espaço alocado para a possibilidade do próximo símbolo ser estimado por uma classe de ordem menor que z_o , será igual a probabilidade de ocorrer um símbolo novo em z_o , e este deverá decrescer conforme mais símbolos ocorrerem em z_o . Cada vez que um símbolo novo ocorrer em alguma classe z_o , as classes de ordens menores que o devem ser consultadas, e, o peso total atribuído a estas deve depender exclusivamente da probabilidade de um novo símbolo ocorrer em z_o . Por exemplo, se a probabilidade de um novo símbolo ocorrer em z_σ é de 0.1, então, z_σ deve ter peso 0.9, e todas as classes de ordem menor que σ devem, juntas, ter peso 0.1.

Denominaremos por *probabilidade de escape* e_o da classe z_o , a probabilidade de um símbolo novo ocorrer em uma classe z_o . Pelos argumentos anteriores, podemos, então, calcular o peso de cada classe conforme sua probabilidade de escape por:

$$u_o(w) = \begin{cases} 1 - e_o, & o = \sigma, \\ (1 - e_o) \prod_{j=o+1}^{\sigma} e_j, & -1 \leq o < \sigma, \end{cases} \quad (2.47)$$

onde σ é a ordem máxima do modelo, e $e_{-1} = 0$, pois, qualquer símbolo pode ser estimado na ordem -1 . Podemos verificar, também, que a soma dos pesos de todas as ordens possui valor 1, o que é esperado ⁸.

A vantagem do uso das probabilidades e_o é que elas são de maior compreensão e visualização do que os pesos u_o , os quais podem se tornar muito pequenos de uma maneira rápida. Veremos, mais adiante, que o mecanismo de calcular probabilidades condicionais usando probabilidades de escape é muito mais fácil de se implementar do que o cálculo a partir da soma ponderada.

A contribuição da probabilidade condicional p_o na probabilidade final pode ser calculada inteiramente usando as probabilidades de escape, isto é:

$$u_o p_o(w) = \prod_{i=o+1}^{\sigma} e_i (1 - e_o) p_o(w). \quad (2.48)$$

A tabela 2.4 mostra um exemplo de um modelo de ordem máxima $\sigma = 4$, sobre o alfabeto $A = \{a, e, i, o, u\}$, onde as probabilidades de escape são obtidas por $1/(C_o + 1)$, e os pesos são obtidos pelo procedimento visto acima.

O método de escape para o cálculo das probabilidades é um mecanismo que resolve o problema da frequência zero. A escolha da probabilidade de escape a ser usada é de fundamental importância na eficiência da compressão, quanto mais esta indicar a chance exata de um símbolo “novo” ocorrer em um certo contexto, maior compressão se poderá obter. Não há, porém, nenhuma base teórica na escolha de uma probabilidade de escape ótima, veremos que uma poderá se comportar melhor para uma certa fonte e pior para outra. Estudaremos nas próximas seções os métodos mais utilizados para o cálculo destas probabilidades, inclusive um novo por nós descoberto.

⁸Para observar que a soma dos pesos vale 1, basta desenvolver a fórmula:
 $\sum_{j=-1}^{\sigma} u_j = (1 - e_{\sigma}) + e_{\sigma}(1 - e_{\sigma-1}) + e_{\sigma}e_{\sigma-1}(1 - e_{\sigma-2}) + \dots + e_{\sigma}e_{\sigma-1}e_{\sigma-2}\dots e_0(1),$
 onde cancelando cada um dos produtos obtemos 1.

ord	Contexto	Cont.	Estimativas - $p_o(w)$ $c_o(w)$						Pesos	Escape
o		C_o	a	e	i	o	u	u_o	e_o	
4	$eaei_-$	0	— 0	— 0	— 0	— 0	— 0	0	1	
3	aei_-	1	0 0	1 1	0 0	0 0	0 0	$\frac{1}{2}$	$\frac{1}{2}$	
2	ei_-	2	$\frac{1}{2}$ 1	$\frac{1}{2}$ 1	0 0	0 0	0 0	$\frac{1}{3}$	$\frac{1}{3}$	
1	i_-	4	$\frac{1}{2}$ 2	$\frac{1}{4}$ 1	$\frac{1}{4}$ 1	0 0	0 0	$\frac{2}{15}$	$\frac{1}{5}$	
0	-	14	$\frac{2}{7}$ 4	$\frac{5}{14}$ 5	$\frac{5}{14}$ 5	0 0	0 0	$\frac{7}{225}$	$\frac{1}{15}$	
-1	-	-	$\frac{1}{5}$ -	$\frac{1}{5}$ -	$\frac{1}{5}$ -	$\frac{1}{5}$ -	$\frac{1}{5}$ -	$\frac{1}{450}$	0	
p^{mista}			$\frac{546}{2250}$	$\frac{1601}{2250}$	$\frac{101}{2250}$	$\frac{1}{2250}$	$\frac{1}{2250}$			

Tabela 2.4: Cálculo das probabilidades mistas com probabilidade de escape $1/(C_o + 1)$ para a palavra “ $aeiiaieiae$ ” sobre o alfabeto $A = \{a, e, i, o, u\}$.

2.4.2 Probabilidade de escape A

Este primeiro método é similar ao método A visto na seção anterior para resolver o problema da frequência zero. Ele se baseia em calcular a probabilidade de escape de tal maneira, que diminua conforme mais símbolos são processados em um certo contexto de ordem o . Logo, e_o^A é obtido pela alocação de uma unidade “extra” sobre o número total de símbolos que já ocorreram em z_o , com isto, temos:

$$e_o^A = \frac{1}{C_o + 1}. \quad (2.49)$$

Por exemplo, se um contexto ocorreu até certo instante 4 vezes (sem contar a última, se ele for o contexto atual), então, a probabilidade de escape será $e_o^A = 1/5$. A probabilidade $p_o^A(w)$ de cada símbolo w (sem levar em conta o escape) é obtida pela relação entre o número de vezes que o símbolo ocorreu dividido pelo número de vezes que o contexto ocorreu:

$$p_o^A(w) = \frac{c_o(w)}{C_o}. \quad (2.50)$$

Se no contexto mencionado o símbolo a ocorreu três vezes, e o símbolo o uma, então, a probabilidade de a é $3/4$ e de o $1/4$. Levando em conta o escape, a probabilidade para cada símbolo w no contexto fica igual à:

$$(1 - e_o^A)p_o^A(w) = \frac{c_o(w)}{C_o + 1}. \quad (2.51)$$

Mais adiante, utilizaremos apenas esta última como a probabilidade para cada símbolo de um contexto, usando o método A. Na tabela 2.4 as probabilidades de escape são calculadas usando este método.

2.4.3 Probabilidade de escape B

Este novo método, apesar do nome, não é o mesmo visto na seção anterior. Ele baseia-se em calcular a probabilidade de escape a partir do número de símbolos distintos que ocorreram em um contexto. Por exemplo, se um contexto de ordem o ocorreu 30 vezes, e destas apenas temos o símbolo a , então, a chance de um símbolo diferente de a ocorrer é pequena; por outro lado, se dentro destas 30 vezes temos 15 símbolos distintos, a probabilidade de ocorrer um símbolo “novo” é provavelmente maior, já que neste caso se deu uma maior variação de símbolos. Podemos, assim, estimar uma probabilidade de escape igual à $1/30$ no primeiro caso, e $15/30$ no segundo. Para a obtenção desta estimativa, o método B usa o procedimento de subtrair uma unidade de cada um dos contadores c_o dos símbolos que aparecem em certo um contexto, sendo que estas unidades serão usadas para a estimativa da probabilidade de escape. Esta subtração pode servir como um filtro a símbolos que ocorrem de modo anômalo em um contexto. Como exemplo, se um contexto aparece 3 vezes, dentre as quais o símbolo a duas e o símbolo o uma, então a probabilidade de a será $(2 - 1)/3 = 1/3$, de o $(1 - 1)/3 = 0$, e de escape $2/3$. Neste caso, somente o próximo o será estimado como um símbolo novo. Logo, a probabilidade de escape e_o^B do método B, é obtida por:

$$e_o^B = \frac{q_o}{C_o}, \quad (2.52)$$

onde, como na seção anterior, q_o indica o número de símbolos distintos que ocorreram no contexto de ordem o . A probabilidade de cada símbolo, sem considerar o escape, fica:

$$p_o^B(w) = \frac{c_o(w) - 1}{C_o - q_o}, \quad (2.53)$$

assim um símbolo não é estimado antes de aparecer duas vezes. Levando em conta o escape, a probabilidade para cada símbolo vale:

$$(1 - e_o^B)p_o^B(w) = \frac{c_o(w) - 1}{C_o}. \quad (2.54)$$

2.4.4 Probabilidade de escape C

Este método é similar ao método B, porém, inicia a contagem dos símbolos a partir de sua primeira ocorrência. Como a probabilidade de escape leva em conta o número de símbolos diferentes em um contexto, esta necessitará ser um pouco menor que a anterior, pois, neste caso, os símbolos possuem maior probabilidade, logo, temos:

$$e_o^C = \frac{q_o}{C_o + q_o}. \quad (2.55)$$

A probabilidade de cada símbolo w , sem considerar o escape, vale:

$$p_o^C(w) = \frac{c_o(w)}{C_o}. \quad (2.56)$$

Levando em conta o escape temos a probabilidade final:

$$(1 - e_o^C)p_o^C(w) = \frac{c_o(w)}{C_o + q_o}. \quad (2.57)$$

2.4.5 Probabilidade de escape R

Este novo método, por nós introduzido, foi obtido da idéia do método B que só leva em conta símbolos que ocorrem mais de uma vez. Desta vez, alocaremos para o escape um espaço proporcional ao número de símbolos que ocorreram no máximo “uma vez” no contexto. Tal mecanismo pode ser claramente justificável, já que símbolos que aparecem mais de uma vez no contexto, não devem ajudar no peso do escape, pois, são símbolos que devem ocorrer novamente no contexto. Definiremos, assim, uma nova função q_o^1 que indica o número de símbolos distintos que ocorreram “uma única vez” no contexto de ordem o . Esta nova função pode ser generalizada por:

$$q_o^r = |w \in A : 0 < c_o(w) \leq r|, \quad (2.58)$$

sendo que $q_o = q_o^\infty$. Como no método A, neste caso adicionaremos uma unidade ao escape, já que todos os símbolos de um contexto podem ter ocorrido mais de uma vez, e sem esta adição, o espaço alocado seria nulo. Este método, inicia a contagem dos símbolos a partir de sua primeira ocorrência. Com isto, a probabilidade de escape e_o^R é calculada por:

$$e_o^R = \frac{q_o^1 + 1}{C_o + q_o^1 + 1}. \quad (2.59)$$

A probabilidade de cada símbolo w , sem considerar o escape, vale:

$$p_o^R(w) = \frac{c_o(w)}{C_o}. \quad (2.60)$$

Levando em conta o escape temos a probabilidade final:

$$(1 - e_o^R)p_o^R(w) = \frac{c_o(w)}{C_o + q_o^1 + 1}. \quad (2.61)$$

Nestes quatro métodos de cálculo da probabilidade de escape, não levamos em conta o fato de que esta probabilidade deve decrescer para um certo contexto, se no mesmo o número de símbolos diferentes estiver muito próximo ao número de símbolos do alfabeto, ou seja, $q_o = m$. De fato, quando usarmos alfabetos grandes, este problema não existirá, pois, poucos contextos terão tal número de símbolos distintos. Já no caso de um alfabeto menor, por exemplo, o alfabeto binário, a maioria dos contextos terá um número de símbolos distintos igual a 2. Para estes casos podemos redefinir o cálculo das probabilidades de escape, devendo, contudo, lembrar da estratégia usada para indicar o final de uma mensagem, vista anteriormente. Se adotarmos um símbolo especial para indicar fim da mensagem, nunca poderemos assumir que algum contexto tenha probabilidade de escape nula, pois, neste caso, podemos estimar este símbolo com probabilidade nula. Sendo assim, com o uso do símbolo especial para fim de mensagem, os métodos vistos funcionam bem, não necessitando da mudança no cálculo. No caso de soubermos o comprimento da mensagem ou no caso de uma transmissão contínua, onde não há a necessidade de um símbolo especial, podemos diminuir consideravelmente, ou até anular, a probabilidade de escape, conforme o número de símbolos distintos se aproximar do tamanho do alfabeto.

2.4.6 Exclusão

No cálculo das probabilidades mistas, verifica-se que o tempo gasto é muito alto, já que para o mesmo são necessárias tanto a obtenção das probabilidades de cada ordem como a soma ponderada das mesmas. Como foi visto no capítulo anterior, o código aritmético necessita para o processamento, além das probabilidades individuais, as acumulativas de cada símbolo, e para as quais o cálculo de todas as individuais faz-se necessário. Além disto, as probabilidades mistas podem ser muito pequenas, acarretando, assim, a necessidade do uso de um codificador aritmético de grande precisão. Sendo assim, o uso de probabilidades mistas é uma técnica não muito prática, apesar de satisfatória do ponto de vista de compressão.

O método de escape pode ser usado como base a uma nova técnica que aproxima o cálculo das probabilidades mistas, e denomina-se *exclusão*. Ela baseia-se em estimar o símbolo dentro do contexto de maior ordem no qual ele possui uma probabilidade não-nula. Por exemplo, se estamos processando um símbolo w usando um modelo de ordem máxima σ , então, o contexto de

ordem σ é o primeiro a ser consultado. Se neste o símbolo w é estimado com probabilidade não-nula, então, somente esta estimativa é usada para codificar w . Caso contrário, envia-se uma probabilidade de escape, e a ordem exatamente inferior é consultada para a estimativa. O mesmo procedimento é executado até que um contexto estime w com probabilidade não-nula. Este processo sempre é finalizado, já que no contexto de ordem -1 todos os símbolos são estimados com probabilidade não-nula. Desta maneira, cada símbolo é codificado por uma seqüência de probabilidades de escape, seguida pela probabilidade do símbolo em um certo contexto.

O termo exclusão é empregado, pois, nesta técnica, todos os contextos de ordem menor ao contexto usado na estimativa são excluídos da probabilidade mista final. Além disto, todos os símbolos que são encontrados em contextos maiores ao usado para a estimativa podem ser excluídos do cálculo da estimativa nos contextos menores, já que estes nunca poderão ser estimados nos mesmos. Isto pode ser executado, assumindo zero aos contadores c_o dos símbolos encontrados em contextos de ordem maior que o . Deve-se salientar, que estes contadores apenas são zerados quando da estimativa de um símbolo em um certo instante deste contexto, voltando ao seu valor original após, já que, em outro instante podem ocorrer contextos de ordens maiores distintos aos atuais. A tabela 2.5 é obtida a partir da tabela 2.4, onde podemos verificar a técnica da exclusão. As entradas em que aparece $\bullet$ indicam que o símbolo foi excluído.

A tabela 2.6 mostra como é codificado cada símbolo do alfabeto para a mesma palavra da tabela 2.5, $\boxed{\text{esc}}$ indica que foi processado um escape. Por exemplo, o símbolo o é codificado por " $\boxed{\text{esc}} \boxed{\text{esc}} \boxed{\text{esc}} \boxed{\text{esc}} \boxed{\text{esc}} i$ " com probabilidades respectivas $1, 1/2, 1/2, 1/2, 1, 1/2$, onde o primeiro $\boxed{\text{esc}}$ é uma transição do contexto $eaei$ para aei , o segundo, de aei para ei , e assim por diante. O símbolo o só é estimado, então, no contexto de ordem -1 . Pode-se verificar que os contextos que não estimam nenhum símbolo possuem probabilidade de escape igual a 1 (no exemplo, os contextos de ordem 4 e 0). As probabilidades de cada símbolo na tabela 2.6 são calculadas assumindo o escape, ou seja, multiplicadas por $(1 - e_o)$.

A exclusão no contexto atual dos símbolos que já ocorreram em contextos de ordens maiores, acarreta em um consumo de tempo considerável, pois, feita esta é necessário recalcular todas as probabilidades dos símbolos restantes, inclusive a de escape. Para resolver este problema, pode-se empregar a técnica de *exclusão parcial*, a qual se comporta da mesma maneira da ex-

ord	Contexto	Cont.	Estimativas - $p_o(w)$ $c_o(w)$						Pesos	Escape
o		C_o	a	e	i	o	u	u_o	e_o	
4	$eaei_-$	0	— 0	— 0	— 0	— 0	— 0	0	1	
3	aei_-	1	0 0	1 1	0 0	0 0	0 0	$\frac{1}{2}$	$\frac{1}{2}$	
2	ei_-	1	1 1	•	0 0	0 0	0 0	$\frac{1}{4}$	$\frac{1}{2}$	
1	i_-	1	•	•	1 1	0 0	0 0	$\frac{1}{8}$	$\frac{1}{2}$	
0	$-$	0	•	•	•	0 0	0 0	0	1	
-1	$-$	—	•	•	•	$\frac{1}{2}$ —	$\frac{1}{2}$ —	$\frac{1}{8}$	0	
p^{mista}			$\frac{1}{4}$	$\frac{1}{2}$	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{16}$			

Tabela 2.5: Cálculo das probabilidades mistas, usando a técnica de exclusão, com probabilidade de escape $1/(C_o+1)$ para a palavra “*aeeiaiaieiae*” sobre o alfabeto $A = \{a, e, i, o, u\}$.

clusão, no sentido de ignorar estimativas de contextos menores ao usado, porém, não excluí símbolos ocorridos em contextos de ordens maiores. Esta técnica têm uma certa perda na compressão (aproximadamente 5%), já que gasta espaço em alguns contextos com símbolos que nunca serão estimados, porém, obtêm uma certa melhora no tempo de processamento. Na prática, o tempo gasto é metade do gasto com uma exclusão total.

Em geral, ao se processar um símbolo w , o modelo é atualizado incrementando-se os contadores $c_o(w)$ para todas as ordens de 0 à ordem máxima σ . Na técnica de exclusão, porém, somente um contexto é usado para a estimativa de w . Como consequência, um novo procedimento, chamado de *contagem parcial*, pode ser adotado. Este procedimento faz com que o contador de um contexto não seja incrementado se o símbolo foi estimado em um contexto de ordem maior, ou seja, o símbolo só é contado no contexto em que ele, realmente, foi estimado. Com isto, os contadores associados a cada contexto indicam o número de vezes que o símbolo foi estimado no mesmo, e não um valor “bruto” (correspondente a todas as vezes que o símbolo ocorreu no contexto ou em contextos de ordens menores). A contagem parcial geralmente melhora a compressão em 2%, e também reduz o tempo de atualização de todos os contadores relacionados, veremos exemplos mais adiante.

Símbolo	Codificação					Código final	
a	<div>esc</div>	<div>esc</div>	a				
	1	$\frac{1}{2}$	$\frac{1}{2}$			$\frac{1}{4} = 2\text{bits}$	
e	<div>esc</div>	e					
	1	$\frac{1}{2}$					$\frac{1}{2} = 1\text{bits}$
i	<div>esc</div>	<div>esc</div>	<div>esc</div>	i			
	1	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$			$\frac{1}{8} = 3\text{bits}$
o	<div>esc</div>	<div>esc</div>	<div>esc</div>	<div>esc</div>	<div>esc</div>	o	
	1	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	1	$\frac{1}{2}$	$\frac{1}{16} = 4\text{bits}$
u	<div>esc</div>	<div>esc</div>	<div>esc</div>	<div>esc</div>	<div>esc</div>	u	
	1	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	1	$\frac{1}{2}$	$\frac{1}{16} = 4\text{bits}$

Tabela 2.6: Exemplo da codificação usando mecanismo de escape com exclusão para a mensagem “*aeiiaiaieiae*” sobre o alfabeto $A = \{a, e, i, o, u\}$.

2.4.7 Divisão de frequências

No capítulo anterior, vimos que o codificador aritmético trabalha com contadores de tamanho fixo para cada símbolo. Nos modelos adaptativos, cada símbolo w possui um contador $c_o(w)$, que indica o número de vezes que w ocorreu no contexto de ordem o . Como usaremos um codificador aritmético para a codificação das estimativas de probabilidades de cada símbolo da mensagem, então, estes contadores devem possuir um tamanho fixo. Sendo assim, a soma dos contadores em um certo contexto não deve ultrapassar um determinado limite estipulado pelo codificador aritmético, definido por *Max_norma*. Existem várias maneiras de manter esta soma dentro do limite *Max_norma*. Uma delas, inclusive a qual usamos em nossa implementação, é dividir todos os contadores relacionados ao contexto por 2, sempre que alguma soma atingir *Max_norma*. Após esta divisão, deve-se tomar o cuidado para que nenhum contador passe a valer 0. Para isso, pode-se arredondar estes para 1 ou eliminar os símbolos associados a estes contadores do contexto.

O mecanismo de manter os contadores com tamanho fixo não é tão desastroso na performance total do processo, isto é, somente pequenos erros

ocorrem na estimativa final da probabilidade. Seja p_b a probabilidade real do símbolo b , de acordo com o modelo usado e ϵ_b o erro devido ao uso de tamanho fixo, logo, o comprimento final por símbolo do código obtido é:

$$\begin{aligned} - \sum_{b \in A} p_b \lg(p_b + \epsilon_b) &= - \sum_{b \in A} (p_b \lg p_b + p_b \lg(1 + \frac{\epsilon_b}{p_b})) \\ &= - \sum_{b \in A} p_b \lg p_b + \frac{- \sum_{b \in A} \epsilon_b + \sum_{b \in A} \frac{\epsilon_b^2}{2p_b} - \sum_{b \in A} \frac{\epsilon_b^3}{3p_b^2} + \dots}{\ln 2}. \end{aligned}$$

Como a soma das probabilidades estimadas valem 1, então $\sum \epsilon_b = 0$, e assumindo que os erros ϵ_b são menores que as probabilidades p_b , então, a perda é aproximadamente da ordem de $\sum \epsilon_b^2$. Logo, se *Max-norma* valer C , os erros são da ordem de $1/C$, e a perda no comprimento médio final da ordem de $1/C^2$.

Na prática observa-se um fato curioso: a divisão de frequência melhora, em vez de piorar, a performance do processo. Este fato entra em total conflito com o fato das probabilidades serem menos exatas quando usamos contadores de tamanho fixo. Este paradoxo pode ser resolvido, a partir da verificação de que em textos usuais as probabilidades não convergem para algum valor exato, ou seja, variam conforme o instante em que ocorrem. Com isto, as estatísticas do começo da mensagem podem ser distintas das do final do texto. Pode-se verificar, por exemplo, neste trabalho, que as palavras mais frequentes variam conforme a seção, podemos ver, por exemplo, que no primeiro capítulo a palavra “aritmética” ocorre um número considerável, já no segundo a palavra “modelo” ocorre mais vezes. Portanto, na prática, estatísticas mais recentes são melhores para as estimativas do que aquelas acumuladas ao longo de todo o processo. A divisão de frequências tem como consequência considerar mais as partes mais recentes da mensagem processada, do que as que ocorreram já a algum tempo, resultando em maior compressão na prática.

Vale a pena observar que nem sempre é válida a melhoria na compressão devido à divisão de frequências. No caso de mensagens geradas por fontes que não possuem variação nas probabilidades ao longo do processo (fontes “fixas”), então a melhor estimativa para a mesma é aquela que obtém probabilidades analisando toda a mensagem já processada, e não apenas a parte

mais recente da mesma. Na prática, como este tipo de mensagem não é muito comum, então a estimativa a partir do uso de divisão de frequências têm bons resultados.

A tabela 2.7 mostra os resultados obtidos por nossa implementação usando probabilidade de escape R, ordem máxima igual a 4, exclusão total e contagem parcial sobre o alfabeto ASCII, com um intervalo de código de 16 bits e uma frequência máxima com valores variando entre 8 (3 bits) e 16384 (14 bits). Os textos contidos nesta tabela fazem parte de um conjunto que é usado como parâmetro no campo das compressões, sendo a descrição de cada um deles feita no apêndice A.

Arquivo	Frequência máxima (<i>Max_norma</i>)						
	8	16	32	64	256	1024	16384
bib	2.1356	1.9947	1.9442	1.9317	1.9288	1.9284	1.9284
book1	2.7311	2.4853	2.3681	2.3174	2.2887	2.2845	2.2837
book2	2.3807	2.1645	2.0698	2.0333	2.0163	2.0145	2.0142
geo	5.3073	4.9990	4.8267	4.7345	4.6778	4.6768	4.6766
news	2.7308	2.5586	2.4818	2.4465	2.4204	2.4135	2.4124
obj1	4.2679	4.0387	3.8988	3.8255	3.7485	3.7199	3.7121
obj2	2.7862	2.6290	2.5585	2.5291	2.5219	2.5192	2.5196
paper1	2.5073	2.4025	2.3732	2.3671	2.3652	2.3652	2.3652
paper2	2.5070	2.3785	2.3355	2.3248	2.3210	2.3206	2.3206
pic	4.2935	3.4859	2.7898	2.1955	1.3481	0.9575	0.8148
progc	2.5518	2.4500	2.4268	2.4226	2.4222	2.4222	2.4222
progl	2.0271	1.8769	1.8156	1.7913	1.7755	1.7726	1.7723
progp	2.1995	2.0284	1.9296	1.8652	1.8006	1.7867	1.7828
trans	1.8407	1.7004	1.6418	1.6172	1.6032	1.6011	1.6008
Média	2.8762	2.6566	2.5329	2.4573	2.3742	2.3416	2.3304

Tabela 2.7: Médias de compressão (bits/char) usando frequência máxima com um número variado de bits.

Podemos notar pela tabela 2.7 que o melhor valor, na prática, para a frequência máxima é geralmente 16384 (14 bits), porém, com pouca variação em relação aos valores menores (15% melhor que o caso de *Max_norma* = 8 e 0.36% que o caso de *Max_norma* = 1024). Esta melhor eficiência para modelos

de frequência máxima maior se deve a existência de uma certa uniformidade das frequências ao longo destes textos.

2.4.8 Tamanho do alfabeto

Em geral um arquivo texto de computador é baseado em bytes (8 bits), logo, o tamanho do alfabeto é de 256 símbolos. Entretanto, existem arquivos, que ao invés de 8 são baseado em números diferentes de bits, por exemplo: um arquivo que contém uma imagem pode ser baseado em 1 bit, e um arquivo contendo números inteiros em 16 bits. Logo, o tamanho do alfabeto é um parâmetro que vale a pena ser analisado. Uma cadeia de 8 bits pode ser dividida em duas cadeias de 4 bits, ou oito de 1 bit. Se mantivermos fixo o comprimento do contexto, teremos uma ordem máxima maior para a mistura de probabilidades. Sendo assim, conseguiremos, em certos casos, melhores resultados. A tabela 2.8 mostra resultados para tamanhos de alfabetos variados, a partir dos mesmos parâmetros da tabela 2.7, sendo que para alfabetos com símbolos de 1 bit, temos ordem máxima igual à 32, para 2 bits ordem de 16, para 3 bits ordem de 12, para 4 ordem de 8, e, acima de 7 ordem de 4.

Pela tabela 2.8 notamos que, na prática, conseguimos maior compressão em modelos com alfabetos de tamanho 256 (8 bits). Isto se deve ao fato de que na maioria dos casos, os textos são organizados em bytes (8 bits). Verificamos que o arquivo “pic” que contém uma imagem, possui pouca variação em seu desempenho para os vários tamanhos de alfabetos. Notamos, também, que os alfabetos de tamanho 16 (4 bits) conseguem uma compressão muito próxima (13% pior) da atingida pelo alfabeto de tamanho 256. Isto porque um símbolo de 8 bits pode ser dividido em dois de 4, e o alfabeto de 8 bits pode ser visto como um alfabeto onde cada símbolo é formado por dois símbolos de 4 bits.

É possível, também, construirmos alfabetos onde cada símbolo não é visto como um simples símbolo, mas como uma concatenação de mais símbolos, sendo denominado de *extensão do alfabeto*. Neste caso o alfabeto pode ser construído adaptativamente, ou seja, crescer conforme a mensagem vai sendo processada. Não trataremos neste trabalho tais casos, nos restringindo no máximo a alfabetos de tamanho 256, pois, como estamos restritos às condições impostas pelo codificador aritmético, onde ao trabalharmos com um espaço de codificação de 16 bits, só podemos calcular frequências de no máximo 14 bits; se usarmos alfabetos maiores, necessitaríamos de frequências

Arquivo	Tamanho do alfabeto					
	1 bit	2 bits	3 bits	4 bits	7 bits	8 bits
bib	5.6861	3.2737	3.4372	2.2690	3.1044	1.9284
book1	4.2819	3.0390	3.3465	2.4823	2.8399	2.2837
book2	4.2749	2.8557	3.1227	2.2302	2.7781	2.0142
geo	13.9374	7.4457	6.3213	5.2788	5.6830	4.6766
news	6.4133	3.8123	3.9204	2.7487	3.5774	2.4124
obj1	10.1265	5.5644	5.2746	4.0874	5.3419	3.7121
obj2	6.9253	4.0288	4.1175	2.8949	4.0270	2.5196
paper1	7.5580	4.2276	4.2926	2.8157	3.9205	2.3652
paper2	6.5913	3.8759	4.0297	2.7103	3.4347	2.3206
pic	1.4829	1.0662	0.9486	0.8677	0.8494	0.8148
progc	7.9733	4.3594	4.4182	2.8893	4.2009	2.4222
progl	5.3243	3.0678	3.3086	2.1074	3.0183	1.7723
progp	5.7545	3.2253	3.4674	2.1559	3.3057	1.7828
trans	5.1011	2.9053	3.2903	1.9546	3.1941	1.6008
Média	6.5308	3.7676	3.8068	2.6780	3.5197	2.3304

Tabela 2.8: Médias de compressão (bits/char) usando alfabeto de tamanho variado.

consideravelmente maiores.

2.4.9 Algoritmos adaptativos

Descreveremos, agora, alguns dos algoritmos existentes baseados em contexto-finito, os quais utilizam as técnicas descritas nesta seção. Eles necessitam de um codificador aritmético (algum dos vistos no capítulo anterior) para codificar as estimativas calculadas por cada um dos modelos, e, todos usam contextos de ordem -1 até uma ordem máxima σ .

Os quatro primeiros algoritmos são denominados pelo prefixo *PPM* (“prediction by partial match”) acrescido da letra que indica o método usado para o cálculo das probabilidades de escape. São eles: PPMA, PPMB, PPMC e PPMC’. Os algoritmos PPMA e PPMB, descritos em [3], usam a técnica de exclusão, e não fazem divisão de frequências (somente quando ocorrer *overflow*), sendo que PPMA utiliza o método A de probabilidade de escape, enquanto PPMB, o método B. O algoritmo PPMC, descrito por Moffat em [13], foi desenvolvido com o objetivo de melhorar a compressão e o tempo de processamento. Para isto, ele utiliza o método C de probabilidade de escape, contagem parcial, e executa divisão de frequências para um espaço de frequências em 8 bits. No mesmo artigo, Moffat descreve um outro algoritmo, denominado por PPMC’, que é semelhante ao PPMC, porém, utilizando exclusão parcial, e assumindo um limite superior para a memória reservada ao modelo, isto é, quando a memória necessária para o modelo for esgotada ele re-inicializa o processo descartando o modelo já construído. Este algoritmo foi desenvolvido com o objetivo de obter mais rapidez e de limitar a memória requerida, e para tal possui uma certa perda na compressão.

Um novo algoritmo, por nós desenvolvido e denominado por PPMR, usa o método R de probabilidade de escape, exclusão total e contagem parcial. Mais adiante, veremos que, na prática, ele é o que obtém melhores resultados.

2.4.10 Implementação dos modelos

Um problema que surge na implementação dos algoritmos baseados em contexto-finito é a implementação dos modelos. Os modelos de contexto-finito de ordem mista, como foi visto até agora, necessitam armazenar, para cada ordem de -1 até a máxima σ , os seus contextos representativos de maneira adaptativa, ou seja, o contexto só será alocado a medida em que ele ocorrer

na mensagem. Além deste armazenamento, deve-se levar em conta o tempo gasto com exclusões, contagens, e cálculo de estimativas.

A implementação que estudaremos, e a qual usaremos, é baseada em árvore. Esta árvore é caracterizada por armazenar em cada nó um símbolo, mais precisamente, um contexto, e o número de vezes em que ele ocorre ao longo do processo. Este tipo de árvore é denominada na literatura por “trie” (ou árvore digital), a qual denominaremos simplesmente por *árvore de contextos*. Cada nível n da árvore de contextos representa o modelo de ordem n , logo, a profundidade da árvore indica a ordem máxima do modelo mais 1, já que o nível 0 representa a ordem -1 .

Em nossa árvore, cada nó é representado por 2 ponteiros: *filho* que aponta para uma cadeia de símbolos que tem como prefixo o contexto que caracteriza este nó, e *prox* que aponta para um próximo contexto de mesma ordem, um identificador *car* que armazena o símbolo do contexto, e um contador *freq* que indica a frequência que este ocorreu ao longo do processo.

O procedimento de construção da árvore é inicializado com um nó *raiz*, que representa o contexto de ordem -1 . A cada passo do processo, existem σ ponteiros, cada um apontando para o contexto atual de cada uma das ordens de -1 a σ . No início, todos estes ponteiros apontam para a raiz, e, a medida que o processo vai sendo executado, cada um deles irá apontar para sua ordem correspondente. A figura 2.3 mostra a árvore de contextos de ordem máxima 2 ao final do processamento da mensagem “aeiiaieieaei”.

O espaço máximo gasto com a árvore de contexto é da ordem de m^σ , onde m é o tamanho do alfabeto e σ é a ordem máxima do modelo. Tal valor pode ser atingido quando, por exemplo, este modelo é aplicado a uma mensagem composta por símbolos aleatórios independentes, e, neste caso todos os contextos poderão possuir todos os m símbolos. Existem casos em que não temos espaço suficiente para a construção completa da árvore de contextos, com isso, devemos estudar uma maneira de limitar o espaço sem que haja muita perda na eficiência do algoritmo. A estratégia descrita em [4] resolve este problema, limitando o espaço reservado para a árvore e fazendo com que o algoritmo não tenha muita perda na eficiência. A estratégia se baseia em armazenar um espaço fixo para os nós da árvore, e, quando o espaço é esgotado, o processo é suspenso e a árvore descartada. Para evitar grandes perdas de eficiência, a medida que a árvore é construída, mantêm-se uma lista circular contendo os últimos símbolos processados. Em [4] o tamanho desta lista é de 2048 símbolos, e é este tamanho que usaremos em nossa implementação.

Figura 2.3: Árvore de contexto após o processamento da mensagem “aeiiaieieaei” para um modelo de ordem máxima 2.

No momento que a árvore é descartada, uma nova árvore de contexto é construída a partir dos símbolos da lista, e o processo é re-iniciado. A estratégia de guardar os últimos símbolos para uma nova inicialização da árvore faz com que esta já esteja adaptada para os últimos símbolos da mensagem, não ocorrendo, assim, perda de eficiência pelo aprendizado da distribuição dos contextos recentes. Para modelos de ordem máxima grande ($\sigma > 4$) não se deve limitar o espaço reservado à árvore em um tamanho pequeno, pois, nestes casos, o espaço se esgotaria um número elevado de vezes tornando, assim, a eficiência bem comprometida.

A tabela 2.9 mostra a compressão obtida usando o algoritmo PPMR com ordem máxima 4, e, limitando-se o espaço reservado à árvore com tamanhos variados. A nossa implementação mantém sempre os nós com tamanho fixo, o qual pode variar de acordo com o máximo de espaço alocado, já que, para espaços menores, podemos utilizar ponteiros de tamanho menor. Verificamos, por esta tabela, que ao usarmos a estratégia em [4], a perda de eficiência é de 16% quando trabalhamos com espaço limitado em 300 Kbytes, o que não é tão desastroso caso haja a necessidade de limitação no espaço.

Uma outra questão a ser analisada é o tempo gasto com a atualização do modelo. Quando um símbolo é processado deve-se verificar a sua ocorrência em cada contexto atual, de ordem 0 à σ , e, incrementar seu contador em contextos que ele ocorre, ou adicioná-lo aos que ainda ele não ocorreu. Para esta simples operação, usando a árvore como foi descrita, gastamos tempo da ordem de σm , onde m é o tamanho do alfabeto. Este tempo pode decrescer à ordem de $\sigma \lg m$ se, ao invés de uma lista simples para representar um contexto, o representarmos em forma de uma árvore de busca binária. Esta representação, porém, fica sem muito sentido no caso de usarmos exclusão total, pois, para este caso, devemos analisar todo o contexto para estimar as probabilidades.

Vejamos o tempo gasto com o cálculo das probabilidades. Esta análise deve levar em conta uma técnica de cada vez. Se a técnica usada for a exclusão total, necessitaremos consultar todo o contexto antes de fazermos a estimativa, ocasionando, com isto, um tempo total da ordem de σm . Contudo, ao usarmos exclusão parcial este tempo pode ser reduzido. Esta redução pode ser feita de várias maneiras, uma delas é colocar sempre o último símbolo lido na frente da lista. Desta forma observa-se, na prática, que o símbolo a ser estimado no contexto aparece, geralmente, no início desta lista. Outra

Arquivo	Tamanho do espaço para a árvore(em Kbytes)					
	100 Kb	200 Kb	300 Kb	600 Kb	1000 Kb	ilimitado
bib	3.2528	2.8379	2.5861	2.1929	1.9284	1.9284
book1	3.6311	3.3222	3.1534	2.8747	2.6780	2.2837
book2	3.1913	2.8750	2.7082	2.4197	2.2603	2.0142
geo	5.2977	5.1653	5.0873	4.9709	4.8895	4.6766
news	3.6406	3.3348	3.1970	2.9630	2.8167	2.4124
obj1	3.8456	3.7790	3.7028	3.7221	3.7121	3.7121
obj2	2.9228	2.7427	2.6738	2.6096	2.5519	2.5196
paper1	3.1637	2.8358	2.6830	2.3896	2.3652	2.3652
paper2	3.2307	2.8940	2.7258	2.4752	2.3206	2.3206
pic	0.8418	0.8268	0.8190	0.8163	0.8146	0.8148
progc	3.0250	2.7550	2.6320	2.4222	2.4222	2.4222
progl	2.0362	1.9094	1.8396	1.7723	1.7723	1.7723
progp	2.0190	1.9192	1.8708	1.7828	1.7828	1.7828
trans	2.5293	2.1245	1.9207	1.6418	1.6008	1.6008
Média	3.0448	2.8087	2.6857	2.5038	2.4225	2.3304

Tabela 2.9: Médias de compressão (bits/char) usando vários tamanhos de espaço para a árvore de contextos.

maneira é manter uma árvore de busca binária para o contexto, tornando o tempo total da ordem de $\sigma \lg m$.

Vimos, anteriormente, que os métodos B, C e R para o cálculo da probabilidade de escape necessitam, além da frequência do símbolo atual, conhecer o número de símbolos distintos de cada contexto, sem o qual a estimativa não pode ser obtida. Mesmo assim, podemos, para estes métodos, manter um contador de símbolos distintos, tornando-os semelhantes ao método A, no qual as estimativas podem ser obtidas apenas pela frequência do contexto, e a frequência do símbolo atual.

Outro tópico a ser analisado é a divisão de frequências, a qual gasta um tempo máximo da ordem de m , pois, quando um contexto atinge frequência máxima, no máximo m símbolos irão ter sua frequência dividida. Como este evento ocorre poucas vezes durante o processamento, não o levaremos em conta na estimativa final do tempo gasto.

Concluindo, para uma mensagem de comprimento t , o tempo total gasto pela atualização do modelo é no mínimo da ordem de $t\sigma \lg m$ (no caso de exclusão parcial, e uso de árvore de busca binária), e no máximo da ordem de $t\sigma m$ (no caso de exclusão total). Em nossa implementação como usaremos exclusão total, o tempo total gasto será da ordem de $t\sigma m$.

2.4.11 Análise comparativa

Para finalizar esta seção, analisaremos a eficiência dos métodos, variando os parâmetros já estudados. Para estas análises usaremos a nossa implementação, o programa “ppm”, que será descrito na próxima seção. Os arquivos usados para estas análises se encontram descritos no apêndice A, e são regularmente usados na análise de técnicas de compressão.

Vejamos, inicialmente, a eficiência obtida ao empregarmos diferentes valores para a ordem máxima do modelo. A tabela 2.10 indica as médias de compressão para um modelo empregando o algoritmo PPMR, para ordem máxima com valores em $\{2, 3, 4, 5, 8, 10\}$. Podemos verificar que a maior eficiência é obtida ao empregarmos uma ordem máxima de 4 ou 5, sendo que valores acima ou abaixo, geralmente, implicam em piores resultados. Isto se deve ao fato de que a maioria dos textos utilizados são formados por palavras padrão⁹ de comprimento com valor médio igual a 4, logo, contextos com

⁹Estas podem ser sílabas, ou palavras reservadas de alguma linguagem computacional.

comprimento próximo deste valor implicam em melhor estimativa da probabilidade. Podemos verificar, ainda, que para o texto “geo” a ordem máxima 2 é a que obtém melhor performance. Isto ocorre devido a este texto ser composto por números de 32 bits, e a maioria destes números possui os 8 bits mais significativos iguais a 0, logo, contextos pequenos (1, 2 ou 3) indicarão este evento muito melhor do que contextos grandes.

Arquivo	Ordem máxima σ do modelo					
	$\sigma = 2$	$\sigma = 3$	$\sigma = 4$	$\sigma = 5$	$\sigma = 8$	$\sigma = 10$
bib	2.6158	2.0836	1.9284	1.9215	2.0117	2.0643
book1	2.8894	2.4404	2.2837	2.2950	2.4658	2.5228
book2	2.8728	2.2436	2.0142	1.9873	2.1105	2.1810
geo	4.5513	4.6510	4.6766	4.6760	4.6748	4.6738
news	3.2313	2.6206	2.4124	2.3969	2.4798	2.5145
obj1	3.9040	3.7422	3.7121	3.7221	3.7500	3.7623
obj2	3.0748	2.7234	2.5196	2.4710	2.4836	2.5096
paper1	2.8820	2.4541	2.3652	2.3911	2.4815	2.5155
paper2	2.8390	2.4114	2.3206	2.3452	2.4633	2.4996
pic	0.8258	0.8169	0.8148	0.8070	0.8169	0.8289
progc	2.8612	2.4842	2.4222	2.4417	2.5199	2.5536
progl	2.3714	1.8900	1.7723	1.7511	1.7735	1.8183
progp	2.2497	1.8384	1.7828	1.7938	1.8593	1.9019
trans	2.3419	1.7651	1.6008	1.5793	1.6023	1.6407
Média	2.8222	2.4403	2.3304	2.3271	2.3924	2.4276

Tabela 2.10: Médias de compressão (bits/char) variando a ordem máxima do modelo.

Uma questão que surge ao estudarmos os métodos de cálculo da probabilidade de escape é identificar aquela que obtém melhores resultados de acordo com a taxa de compressão. A tabela 2.11 indica a taxa de compressão para os algoritmos PPMA, PPMB, PPMC e PPMR. Todos utilizam exclusão total, ordem máxima 4, e, com exceção do PPMB que utiliza contagem total¹⁰,

¹⁰O PPMB necessita de contagem total, pois sem ela não obtém compressão, já que neste caso, os únicos símbolos que serão estimados são símbolos da ordem -1, e com isso as outras ordens nunca terão símbolos com frequência maior que 1.

os outros utilizam contagem parcial. Pela mesma tabela podemos notar que o algoritmo PPMR é o que obtém os melhores resultados na maioria dos casos, sendo pior apenas em arquivos que possuem tamanho relativamente pequeno. O algoritmo PPMC atinge bons resultados, sendo muitas vezes bem próximos ao PPMR. Os algoritmos PPMA e PPMB, são os que atingem os piores resultados, sendo na maioria dos casos o PPMA melhor que o PPMB.

Arquivo	Algoritmos adaptativos			
	PPMA	PPMB	PPMC	PPMR
bib	2.0026	2.1667	1.9395	1.9284
book1	2.3332	2.3610	2.3303	2.2837
book2	2.0682	2.1721	2.0417	2.0142
geo	6.0834	4.6810	4.7223	4.6766
news	2.5815	2.6967	2.4281	2.4124
obj1	4.5179	4.2902	3.7314	3.7121
obj2	2.8109	2.9011	2.5103	2.5196
paper1	2.4800	2.6695	2.3696	2.3652
paper2	2.4087	2.5012	2.3517	2.3206
pic	0.8849	0.8482	0.8295	0.8148
progc	2.5934	2.7738	2.4112	2.4222
progl	1.8162	2.0959	1.7693	1.7723
progp	1.8458	2.0960	1.7637	1.7828
trans	1.6433	1.9657	1.5827	1.6008
Média	2.5764	2.5871	2.3415	2.3304

Tabela 2.11: Médias de compressão (bits/char) variando o algoritmo adaptativo com $\sigma = 4$.

Uma outra análise que podemos fazer é verificar a variação da performance ao usarmos, contagem total ou parcial, exclusão parcial ou total. A tabela 2.12 mostra a taxa de compressão para o algoritmo PPMR, com ordem 4, para exclusão parcial e total, e contagem parcial e total. Pela tabela podemos verificar que ao utilizarmos exclusão total temos um ganho em média de 8%, e, para contagem parcial, de 5%.

Vimos anteriormente que uma maneira de resolver o problema do com-

Arquivo	Parâmetros do algoritmo			
	E.P., C.P.	E.T., C.T.	E.P., C.T.	E.T., C.P.
bib	2.1032	2.0301	2.3130	1.9284
book1	2.4335	2.3077	2.5529	2.2837
book2	2.1527	2.0816	2.3117	2.0142
geo	5.1638	4.8265	5.3938	4.6766
news	2.6237	2.5303	2.8617	2.4124
obj1	3.9617	4.0588	4.4449	3.7121
obj2	2.7028	2.6859	2.9918	2.5196
paper1	2.6186	2.4719	2.8600	2.3652
paper2	2.5628	2.3816	2.7505	2.3206
pic	0.8787	0.8844	0.9992	0.8148
progc	2.6706	2.5579	2.9370	2.4222
progl	1.9257	1.9142	2.1702	1.7723
progp	1.9331	1.8874	2.1465	1.7828
trans	1.7119	1.7198	1.9180	1.6008
Média	2.5316	2.4527	2.7608	2.3304

Tabela 2.12: Médias de compressão (bits/char) usando PPMR de ordem 4, variando os parâmetros: E.P.- exclusão parcial, E.T.- exclusão total, C.T.- contagem total, C.P.- contagem parcial.

primário do contexto a ser utilizado pelos modelos de contexto-finito é empregar mistura de contextos. Contudo, não verificamos, ainda, o quanto a performance do algoritmo é melhorada ao empregarmos este tipo de técnica. A figura 2.4 nos mostra o comportamento do algoritmo PPMR, com ordem máxima 4, exclusão total e contagem parcial em relação ao número de símbolos processados ao longo do arquivo “book1”. A linha tracejada indica a compressão obtida sem o uso de probabilidades mistas, isto é, se o símbolo não for estimado no contexto de ordem máxima (no exemplo, 4), então ele é estimado no contexto de ordem -1 . A linha cheia corresponde a compressão obtida com o uso de probabilidades mistas. Pela figura, notamos que no início do processo o uso de probabilidade mista acarreta em um ganho considerável na compressão, já que a maioria dos símbolos são estimados em contextos de ordens inferiores. Após um grande número de símbolos processados, a maioria dos símbolos serão estimados no contexto de ordem máxima, fazendo com que a diferença entre as duas técnicas seja reduzida. Sendo assim, para arquivos grandes com tamanho proporcional ao número de símbolos do alfabeto multiplicado pelo número de classes de equivalência total do modelo, o uso de probabilidade mista não acarreta em grande contribuição na taxa de compressão do algoritmo.

2.5 O programa ppm

Nossa implementação, denominada por *ppm*, é baseada em modelos de contextos finitos usando codificação aritmética, e pode utilizar como probabilidade de escape os métodos A, B, C ou R. Ela possui a capacidade de empregar contagem parcial ou total, exclusão parcial ou total, probabilidade mista ou simples, podendo variar a ordem máxima, definir o espaço máximo a ser alocado para a árvore de contextos e a frequência máxima de cada símbolo. Como foi observado na seção anterior, o método R de ordem 4, com exclusão total e contagem parcial, é o que obtém os melhores resultados ao longo dos testes. Sendo assim, o algoritmo possui como valores iniciais estes parâmetros, os quais podem ser alterados na chamada do mesmo. Este programa é público, e para maiores detalhes favor me contactar em (mris@ime.usp.br).

Todos os resultados da seção anterior foram obtidos usando o algoritmo “ppm”. A tabela 2.13 foi extraída parcialmente de [19]. Ela compara diferen-

Figura 2.4: Taxa de compressão usando PPMR com ordem 4, durante o processo do arquivo “book1”, empregando probabilidade mista (linha cheia) ou simples (linha tracejada).

tes algoritmos de compressão, indicando a taxa de compressão que cada um obtém para os mesmos arquivos já analisados na seção anterior. Os primeiros três algoritmos são variações do algoritmo de Ziv e Lempel, que são baseados em dicionário. Uma descrição completa destes métodos pode ser encontrada no capítulo 8 de [19]. O “compress” do Unix usa um algoritmo desta classe de algoritmos. “HUFF” é o algoritmo baseado no método de Huffman. “GZIP” é um programa de compressão, muito utilizado ultimamente e que obtém resultados muito satisfatórios, tanto do ponto de vista de compressão, como com de tempo. “PPMR4” é o nosso algoritmo, usando o método de probabilidade de escape R com ordem máxima 4. Pela tabela notamos que nosso algoritmo obtém os melhores resultados, resultando em média uma melhora de 14% em relação ao “gzip”.

Arquivo	Algoritmos					
	LZ77	LZ78	LZFG	HUFF	GZIP	PPMR4
bib	3.75	3.95	2.90	5.24	2.50	1.9284
book1	4.57	3.92	3.62	4.56	3.25	2.2837
book2	3.93	3.81	3.05	4.83	2.70	2.0142
geo	6.34	5.59	5.10	5.70	5.34	4.6766
news	4.37	4.33	3.44	5.23	3.06	2.4124
obj1	5.41	5.58	4.03	6.06	3.83	3.7121
obj2	3.81	4.68	2.96	6.30	2.63	2.5196
paper1	3.94	4.50	3.03	5.04	2.79	2.3652
paper2	4.10	4.24	3.16	4.65	2.89	2.3206
pic	2.22	1.13	0.87	1.66	0.82	0.8148
progc	3.84	4.60	2.89	5.26	2.68	2.4222
progl	2.90	3.77	1.97	4.81	1.80	1.7723
progp	2.93	3.84	1.90	4.92	1.81	1.7828
trans	2.98	3.92	1.76	5.58	1.61	1.6008
Média	3.94	4.13	2.95	4.99	2.69	2.3304

Tabela 2.13: Médias de compressão (bits/char) obtida para alguns algoritmos

Neste trabalho só nos preocupamos com a eficiência dos algoritmos do ponto de vista de taxa de compressão, não levando em conta o tempo gasto com a mesma. A partir destes testes chegamos ao resultado que o tempo

médio gasto com a compressão a partir do “ppm” é cerca de dez vezes o tempo gasto pelo “gzip”. Isto se deve ao fato dos inúmeros parâmetros usados pelo nosso programa, os quais necessitam de muitas análises ao longo do processamento. O programa foi desenvolvido desta maneira com o objetivo de facilitar a análise, do emprego ou não, destes mesmos parâmetros.

Várias mudanças são propostas para um próximo trabalho ao “ppm”, vamos enumerar algumas:

- melhora no tempo de processamento a partir de um novo mecanismo de atualização dos modelos.
- processamento por blocos, isto é, dividir a mensagem em blocos e processá-las independentemente para ser possível a recuperação da mensagem, ou parte dela, mesmo se o código estiver danificado (transmissão com ruídos).
- utilização de outros codificadores, além do aritmético, tais como Huffman adaptativo.
- melhoria na estrutura de armazenamento do modelo, tentando reduzir o espaço necessário para o mesmo.
- análise da utilização dos métodos vistos para a compressão de imagens.

Apêndice A

Conjunto de arquivos

As análises feitas ao longo deste trabalho, usa um conjunto de arquivos denominados na literatura por *Compression Corpus*. Neste apêndice, mostraremos as características de cada um destes arquivos.

bib : arquivo ASCII, formato “Unix refer”, que contém 725 referências de livros e trabalhos em Ciência da Computação.

Tamanho : 111,261 bytes.

book1 : arquivo ASCII, não formatado, que contém o livro “Far from the Mad-ding Crowd” de Thomas Hardy.

Tamanho : 768,771 bytes.

book2 : arquivo ASCII, formato “Unix troff”, que contém o livro “Principles of Computer Speech” de Ian Witten.

Tamanho : 610,856 bytes.

geo : arquivo de números de 32 bits, que contém dados sísmicos.

Tamanho : 102,400 bytes.

news : contém “eletronic news”.

Tamanho : 377,109 bytes.

obj1 : arquivo executável para o VAX, contém a compilação do programa “progp”.

Tamanho : 21,504 bytes.

obj2 : arquivo executável para o Apple-Macintosh.

Tamanho : 246,814 bytes.

paper1 : arquivo ASCII, formato "Unix troff", que contém o trabalho "Arithmetic coding for data compression" - Witten, Neal e Cleary.

Tamanho : 53,161 bytes.

paper2 : arquivo ASCII, formato "Unix troff", que contém o trabalho "Computer (In)security" - Witten.

Tamanho : 82,199 bytes.

pic : figura com 1728x2376 "bit map" e 200 ppi, formato "CCITT facsímile".

Tamanho : 513,216 bytes.

progC : código fonte escrito na linguagem "C", formato ASCII, contém o programa "compress-versão 4.0" do Unix.

Tamanho : 39,611 bytes.

progl : código fonte escrito em "LISP", formato ASCII.

Tamanho : 71,646 bytes.

progp : código fonte escrito na linguagem "PASCAL", formato ASCII, contém um programa para verificar a eficiência do PPM.

Tamanho : 93,695 bytes.

trans : transcrição do controle do "EMACS" sobre o terminal.

Tamanho : 93,695 bytes.

Índice

- alfabeto, 6
- algoritmos adaptativos, 86
- árvore de Huffman, 8-9
 - construção, 8
- árvore digital, 87
- árvore de contextos, 87
- bit-stuffing, 35
- $c^z(M)$, 55
- c_o , C_o , 71
- classes de equivalência, 50-52, 71
- codificação aritmética, 4-5, 12
 - algoritmo básico, 16
 - alfabeto binário, 34
 - alfabeto não-binário, 37
 - com o uso de divisões, 5
 - eficiência da implementação, 28, 40
 - exemplo de codificação, 12
 - exemplo de decodificação, 12
 - história, 4
 - implementações, 19
 - problemas na implementação, 16
 - sem o uso de divisões, 5, 31
 - tempo gasto, 31
- codificação e decodificação incrementais, 18
- código compacto, 4
- código de Elias, 5
- código de Huffman, 8
 - otimalidade do, 9
- código em bloco, 4
- código em não-bloco, 4
- código enumerativo, 5, 55
- código instantâneos, 4
- código não-singular, 4
- código ótimo, 7
- comprimento ideal do código, 29
- comprimento médio ótimo, 43
- contagem parcial, 80
 - análise, 93
- $d^{zx}(y)$, 55
- decodificação
 - problema da, 21
- divisão de frequências, 23, 81
 - tempo gasto com, 91
- e_o , 72
- e_o^A , 74
- e_o^B , 75
- e_o^C , 76
- e_o^R , 77
- e_{-1} , 73
- eficiência em tempo, 89
- entropia, 7
 - de um modelo, 48
- exclusão, 78-79
 - análise, 93

- parcial, 79, 89
- total, 89
- extensão do alfabeto, 4, 84
- fonte de informação, 6, 46
- história de uma mensagem, 50
- instância, 6
- lista circular, 87
- máquina de estados finitos, 47
- Max-norma*, 27, 81
- mensagem, 6
 - menor, 32
- métodos adaptativo
 - A, 64
 - B, 65
- método aproximado, 57
- método exato, 56
- método fixo, 59
- mistura de estimativas, 71
- mistura de ordens, 70
- mistura de probabilidades, 78
 - análise, 95
- modelo, 46
 - adaptativo, 50, 63
 - adaptativo de contexto-finito, 70
 - comparação entre adaptativo e semi-adaptativo, 65-69
 - de contexto-finito, 47
 - de contexto-finito caracterizado por estados finitos, 47
 - de estados finitos, 47
 - fixo, 7, 9
 - implementação, 86
 - limitando o espaço gasto, 87
 - não-adaptativo ou estático, 49
 - ordem de um, 47
 - semi-adaptativo, 49, 53
 - semi-adaptativo (comparações entre), 60-62
- normalização das probabilidades, 19, 33
- ordenação instantânea, 6
- ordem máxima
 - análise, 91
- overflow, 23, 35
- palavra, 6
- palavra-código,
 - comprimento da, 12
- PPMA, 86
- PPMB, 86
- PPMC, 86
- PPMC', 86
- PPMR, 86
- precisão finita, uso de, 29
- probabilidade
 - adaptativa, 64
 - adaptativa A, 64
 - adaptativa B, 65
 - aproximada, 59
 - condicional, 50, 52, 56
 - de escape, 72
 - de escape (análise), 92
 - de escape A, 74
 - de escape B, 75
 - de escape C, 76
 - de escape R, 77
 - exata, 56
 - mista, 71
 - fixa, 59
 - real, 46

problema de fim de mensagem, 13,
29, 52, 78
problema da frequência zero, 64, 70,
72
processo incremental, 18, 33
programa ppm, 91, 95
propriedade dos prefixos, 4
 q^z , 64
 q_o^r , 77
quantidade de informação, 6
recálculo de frequências, 31
símbolos aleatórios independentes,
87
tabela de resultados, 97
tamanho do alfabeto, 84
transmissões contínuas, 15, 78
trie, 87
 u_o , 72
underflow, 23

Bibliografia

- [1] N. Abramson. *Information Theory and Coding*. McGraw-Hill Book Co., Inc., New York, second edition, 1963.
- [2] J.G. Cleary and I.H. Witten. A comparison of enumerative and adaptive codes. *IEEE Trans. Information Theory*, IT-30:306–315, March 1984.
- [3] J.G. Cleary and I.H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Trans. on Communications*, 32(2):396–402, April 1984.
- [4] G.V. Cormack and R.N.S. Horspool. Data compression using dynamic markov modelling. *The Computer Journal*, 30(6):541–550, December 1987.
- [5] T. M. Cover. Enumerative source encoding. *IEEE Trans. Information Theory*, IT-19(1):73–77, 1973.
- [6] M. Guazzo. A general minimum-redundancy source-coding algorithm. *IEEE Trans. Information Theory*, IT-16(1):15–25, January 1980.
- [7] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proc. Inst. Electr. Radio Eng.*, 40(9):1098–1101, 1952.
- [8] J. G. Cleary I. H. Witten and R. M. Neal. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, June 1987.
- [9] G. G. Langdon. Tutorial on arithmetic coding. Technical Report RJ3128, IBM Research Laboratory, San Jose, CA, 1981.
- [10] G. G. Langdon. An introduction to arithmetic coding. *IBM J. Research and Development*, 28(2):135–149, March 1984.

- [11] G.G. Langdon and J. Rissanen. Universal modeling and coding. *IEEE Trans. Information Theory*, IT-27:12–23, January 1981.
- [12] G.G. Langdon and J. Rissanen. A simple general binary source code. *IEEE Trans. Information Theory*, IT-28:800–803, September 1982.
- [13] Alistair Moffat. Implementing the ppm data compression scheme. *IEEE Trans. on Communications*, 38(11):1917–1921, November 1990.
- [14] R. Pasco. *Source coding algorithms for fast data compression*. PhD thesis.
- [15] J. Rissanen and K.M. Mohiuddin. A multiplication-free multialphabet arithmetic code. *IEEE Trans. on Communications*, 37(2):93–98, February 1989.
- [16] J. J. Rissanen. Generalized kraft inequality and arithmetic coding. *IBM J. Research and Development*, 20:198–203, May 1976.
- [17] J. J. Rissanen and G. G. Langdon. Arithmetic coding. *IBM J. Research and Development*, 23(2):149–162, March 1979.
- [18] F. Rubin. Arithmetic stream coding using fixed precision registers. *IEEE Trans. Information Theory*, IT-25(6):672–675, November, 1979.
- [19] J.G. Cleary T.C. Bell and I.H. Witten. *Text Compression*. Prentice Hall, Englewood Cliffs, New Jersey 07632, 1990.