MARCOS ALEXANDRE ARAÚJO SIQUEIRA

PROTEÇÃO DESIGUAL DE ERROS EM ARQUIVOS COMPACTADOS

Dissertação apresentada ao Programa de Pós-Graduação em Informática Aplicada da Pontifícia Universidade Católica do Paraná como requisito parcial para obtenção do título de Mestre em Informática Aplicada.

CURITIBA

2005

MARCOS ALEXANDRE ARAÚJO SIQUEIRA

PROTEÇÃO DESIGUAL DE ERROS EM ARQUIVOS COMPACTADOS

Dissertação apresentada ao Programa de Pós-Graduação em Informática Aplicada da Pontifícia Universidade Católica do Paraná como requisito parcial para obtenção do título de Mestre em Informática Aplicada.

Área de Concentração: *Redes de Computadores e de Telecomunicações*

CURITIBA

2005

Siqueira, Marcos Alexandre Araújo
Proteção desigual de erros em arquivos compactados. Curitiba, 2005. 109p.

Dissertação (Mestrado) – Pontifícia Universidade Católica do Paraná. Programa de Pós-Graduação em Informática Aplicada.

1.LZSS 2. LZ77 3. Proteção Desigual de Erros 4.Compressão de Dados 5. Codificação de Canal I.Pontifícia Universidade Católica do Paraná. Centro de Ciências Exatas e de Tecnologia. Programa de Pós-Graduação em Informática Aplicada II-t

Esta página deve ser reservada à ata de defesa e termo de aprovação que serão fornecidos pela secretaria após a defesa da dissertação e efetuadas as correções solicitadas.

À minha família

Agradecimentos

Ao professor orientador Dr. Marcelo Eduardo Pellenz, pela orientação valiosa e precisa.

À minha esposa e filha pelo apoio e compreensão pelas horas de convivência que lhes foram diminuídas.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Símbolos

C	Código de Bloco
d_{MIN}	distância mínima
F	Field
G	Grupos
GF(q)	Galois field
p	probabilidade
q	quantidade de caracteres
u	Seqüência de informação
v	Seqüência discreta codificada chamada palavra de código
r	Seqüência recebida **r**
û	Seqüência binária ou seqüência recebida decodificada
o_i,f_i,u_i	Itens do token (tripla)
X	Entrada de dados
X_C	Dados compactados
Y	Reconstrução dos dados

Lista de Abreviaturas

ADSL	*asymmetric digital subscriber line*
AEC	*average error cost*
ARQ	*Automatic Repeat Request*
ASCII	*American Standard Code for Information Interchange*
ASIC	*Applicatios Specific Integrated Circuit*
BCH	*Bose Chaudhuri Hoquingham Code*
BER	*Bit Error Ratio*
CDMA	*Code Division Multiple Access*
CRC	*CyclicRedundance Check*
ECC	*Error Correction Code*
FEC	*Forward Error Correction*
FPGA	*Field Programmable Gate Array*
HTML	*Hipertext Mark Language*
LZSS	*Lempel –Ziv Storer and Szymanski algorithm published in 1982*
LZ77	*Lempel –Ziv algorithm published in 1977*
LZ78	*Lempel –Ziv algorithm published in 1978*
LZW	*Lempel –Zivand Welch algorithm*
RS	*Reed-Solomon*
SEC – B_l EL	*Single-bit Error Correcting and l-bit burst Error Locating Codes*
UEP	*Unequal Error Protection*
VHDL	*Very high speed integrated circuit Hardware Description Language*

Resumo

Os algoritmos de compressão sem perdas baseados na codificação de Lempel-Ziv são amplamente usados para compressão de textos e também em determinados esquemas de compressão com perdas para imagem e vídeo. Em sistemas multimídia, sempre é necessário manipular, armazenar e transmitir um grande volume de dados compactados. Contudo, dados compactados são muito sensíveis a qualquer evento de erro, devido ao fato que estes erros podem se propagar durante o processo de descompressão. Portanto, a utilização de técnicas de codificação de canal otimizadas para transmissão e armazenagem de dados digitais compactados através de canais ruidosos pode minimizar significativamente estes efeitos. Neste trabalho propomos uma estratégia de codificação de canal para minimizar o impacto de surtos de erros na transmissão ou armazenagem de arquivos compactados. A estratégia proposta é para o algoritmo de compressão baseado em dicionário dinâmico LZSS, amplamente utilizado para compressão de dados. Investigamos através de uma extensa análise computacional as classes de informação mais sensíveis a erros e aplicamos uma estratégia para proteção desigual que minimiza a propagação de erros no processo de descompressão.

Palavras-Chave: LZSS, LZ77, proteção desigual de erros, compressão de dados, codificação de canal.

Abstract

Lossless compression algorithms based on Lempel-Ziv coding are widely used for text compression and for certain audio and video lossy compression schemes, as well. On multimedia systems, it is always necessary to manipulate, transmit and store a large compressed data volume. However, compacted data are highly sensitive to any error event, due to error propagation possibilities during the decompression process. As a result, the usage of optimized channel coding techniques for digital compressed data transmission and storage through noisy channels can abbreviate significantly those effects. In this study a channel coding strategy was proposed to minimize the burst of errors impact over compacted files transmission and storage. The suggested scheme is directed to a dynamic dictionary based compression LZSS algorithm, frequently used on data compression. An extensive computing analysis of the most error sensitive information classes was performed and an unequal protection strategy that minimizes error propagation in the decompression process was applied.

Key-words: LZSS, LZ77, unequal error protection, data compression, channel coding.

Capítulo 1

Introdução

Nos anos 40, os primeiros anos da teoria da informação, a idéia de desenvolver novas técnicas de codificação estava se formando. Pesquisadores exploravam as idéias de entropia, conteúdo de informação e redundância. Uma noção popular residia em que se a probabilidade de símbolos em uma mensagem era conhecida, deveria haver uma maneira de codificar os símbolos de tal forma que a mensagem ocuparia menos espaço. Nos anos 90, parecia natural que a teoria de informação estivesse popularizada com a programação de computadores, mas logo após a 2ª Guerra Mundial, para todos os propósitos práticos, não havia computadores digitais. Então a idéia de desenvolver algoritmos usando a base 2 para codificação de símbolos foi considerado um grande passo [NEL91].

Analisando as técnicas de compressão disponíveis na literatura específica, selecionou-se em apresentar algumas aplicações que serviram de motivação para este estudo. Cada uma dessas aplicações diferencia-se pela sua utilização. Tem-se então:

- Aplicações voltadas para a transmissão em redes cabeadas (Internet);
- Aplicações visando a transmissão sem fio;
- Aplicações da compressão sem perdas para a armazenagem de dados.

Entre as aplicações voltadas para a transmissão em redes cabeadas, pode-se citar o aumento da velocidade da rede através da compressão do http e a compressão adaptativa dos dados, que atualiza o nível de compressão de acordo com o tamanho da fila de saída [KRA00].

Entre as aplicações em transmissão sem fio estão a recuperação de erro para códigos de comprimento variável, os códigos de correção de erro e de compressão combinados [FUJ00].

Entre as aplicações da compressão sem perdas para a armazenagem de dados, está o algoritmo de compressão estatística de Lempel-Ziv ou LZ77, descrito como um esquema universal de código que pode ser aplicado a qualquer fonte discreta [ZIV77], e do qual derivam os algoritmos LZ78 [ZIV78], LZW [SAY00] e LZSS [SAY00], entre outros.

1.1 Desafio

O primeiro método mais popular de codificar símbolos eficientemente é agora conhecido como codificação de Shannon-Fano. Claude Shannon nos laboratórios Bell e R.M.Fano no M.I.T. (Massachussets Institute of Tecnology) desenvolveram este método quase simultaneamente. Ele depende de se conhecer simplesmente a probabilidade de ocorrência de um símbolo aparecer na mensagem [NEL91] .

Dadas as probabilidades, a tabela de códigos poderia ser construída de forma a possuir as seguintes importantes propriedades:

- Códigos diferentes possuem diferentes números de bits.
- Códigos para símbolos com baixa probabilidade possuem mais bits, e códigos para símbolos com alta probabilidade possuem poucos bits.
- Portanto os códigos são de comprimentos diferentes de bits, e eles podem ser codificados de forma única. [NEL91]

As primeiras duas propriedades se aplicam indistintamente. O desenvolvimento de códigos que variam no comprimento de acordo com a probabilidade dos símbolos que estão sendo codificados por tais códigos, torna possível a compressão de dados. [NEL91]

As técnicas de compressão de dados podem ser divididas em duas grandes famílias: com e sem perdas. Compressão de dados *com perdas* permite certa perda de precisão em troca de um incremento na taxa de compressão. Compressão com perdas prova ser efetiva quando aplicada a imagens gráficas ou voz digitalizada. Pela própria natureza, estas representações digitalizadas de fenômenos analógicos não são perfeitas, então a idéia da entrada e saída não possuírem perfeita correspondência é um pouco mais aceitável. Muitas técnicas de compressão com perdas podem ser ajustadas para diferentes níveis de qualidade, ganhando maior precisão em troca de uma compressão menos efetiva. Até recentemente, a compressão de dados com perdas pode ser implementada usando hardware dedicado. Nos últimos anos, os

mais poderosos programas de compressão com perdas foram transferidos para computadores, mas mesmo assim, o campo ainda é dominado por implementações de hardware. [NEL91]

A compressão sem perdas consiste na técnica que garante a geração de duplicatas exatas dos dados de entrada, depois do ciclo de compactação/ descompactação. Este é o tipo de compressão usada para o armazenamento de registros de base de dados e para arquivos de texto. Nestas aplicações, a perda de um único bit pode ser catastrófica. Entre as técnicas sem perdas estão os algoritmos estatísticos, como, por exemplo, o *Algoritmo de Huffman* e os algoritmos baseados em dicionário, como LZ77, LZ78, LZW e LZSS. No contexto das comunicações, a teoria da informação provê respostas a duas questões fundamentais:

- Qual é a condição extrema sob a qual um sinal não pode ser compactado?

- Qual é a mínima taxa de transmissão para a comunicação confiável através um canal com ruído?

As respostas a estas duas questões, entre outras pesquisadas por Shannon, se baseiam na entropia da fonte e na capacidade do canal, respectivamente. A "Entropia" é definida em termos de comportamento probabilístico de uma fonte de informação, enquanto que a "Capacidade" é definida como a habilidade intrínseca de um canal em transportar a informação. Esta última está relacionada com as características de ruído do canal [PRO95].

A codificação fonte é utilizada para remover a redundância descontrolada que ocorre naturalmente em algumas fontes de informação, favorecendo a redução da taxa de transmissão de um dado sistema. Por exemplo, em um arquivo texto, certos caracteres estarão repetidos, da mesma forma que na transmissão de voz e de vídeo. Codificar a fonte significa compactar a informação, por meio da eliminação da redundância descontrolada, ou seja, sem padrão. A codificação de canal é utilizada na informação transmitida para aumentar a imunidade do sinal ao ruído do canal, por meio da inserção de redundância controlada. Blocos de informação, ou seqüência de bits, são adicionados à informação compactada, visando a detecção e correção de erros. A capacidade de correção de erros está diretamente relacionada com a complexidade do código utilizado [PEL04]. A redução na taxa de transmissão de um dado sistema requer o uso de algoritmos de compressão, podendo estes serem com ou sem perdas. Através da teoria da informação, constatou-se que se a entropia da fonte for menor do que a capacidade do canal, pode-se conseguir a comunicação sem erros sob aquele canal.

A transmissão e o armazenamento de informação digital têm muito em comum. Ambos transferem dados de uma fonte de informação para um destino (ou usuário). Um sistema típico de transmissão pode ser representado pelo diagrama em blocos abaixo:

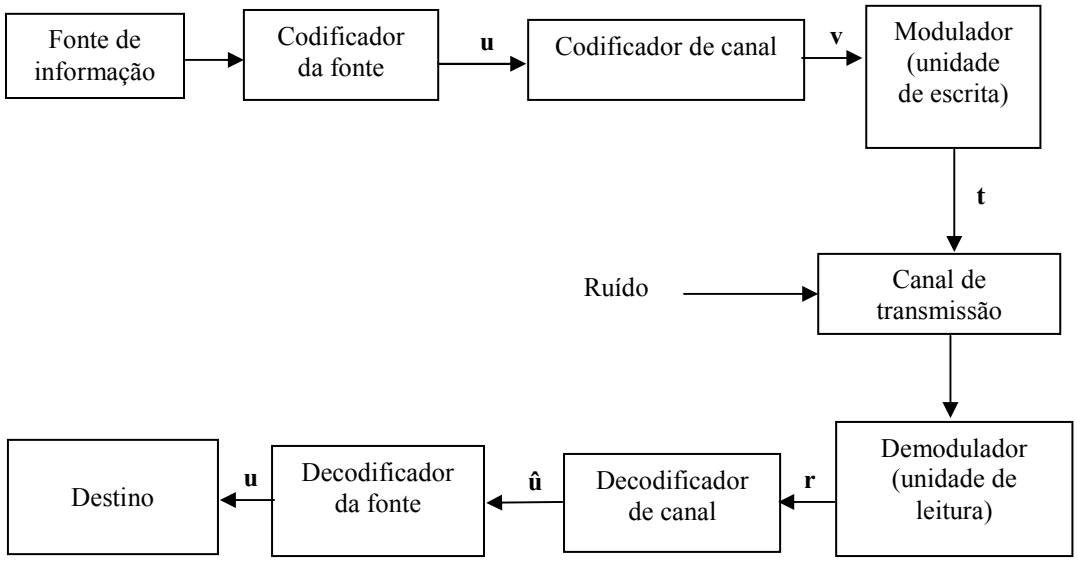

Fig 1.1. Diagrama em blocos de um típico sistema de transmissão digital [LIN83]

A Figura 1.1 apresenta os blocos funcionais de um sistema típico de transmissão digital. O codificador da fonte transforma a saída da fonte de informação em uma seqüência de bits, chamada seqüência de informação **u**. O codificador de canal transforma a seqüência **u** em uma seqüência discreta **v** chamada palavra código (code word). O modulador transforma cada símbolo de saída do codificador de canal em uma forma de onda de duração **t** segundos adequada para transmissão. Esta forma de onda entra no canal de transmissão e está suscetível ao ruído. O demodulador processa cada forma de onda de duração **t** e produz uma saída que pode ser discreta (quantizada) ou contínua. A seqüência da saída do demodulador correspondente à seqüência codificada **v** é chamada de seqüência recebida **r**. O decodificador de canal transforma a seqüência recebida **r** em uma seqüência binária **û** chamada de seqüência estimada. A estratégia do decodificador está baseada em regras para o codificador de canal e das características do ruído do canal de transmissão. Idealmente, **û** será uma réplica da seqüência de informação **u**, embora o ruído possa causar alguns erros de decodificação [LIN83].

Para assegurar maior confiabilidade na transmissão dos dados compactados, novas técnicas têm sido desenvolvidas para o tratamento conjunto da compressão e da codificação de canal. Este estudo tem como desafio analisar o impacto dos surtos de erro sobre o algoritmo de compressão LZSS e através dos resultados obtidos, propor um esquema de proteção desigual de erros para as classes mais importantes dos dados compactados. Mais ainda, adotar para os parâmetros associados ao código de proteção desigual de erros, Reed-Solomon, valores que permitam a detecção e a correção desses surtos de erro de forma eficaz sem comprometer a desempenho do código de compressão empregado, no caso o LZSS.

1.2 Motivação

Sabe-se que uma grande variedade de códigos de compressão é utilizada continuamente nas mais diferentes aplicações, buscando-se melhor aproveitamento da largura de banda disponibilizada e maior otimização de espaço em disco ou de memória. Quanto ao aproveitamento da largura de banda, observa-se a crescente demanda do acesso a diferentes serviços que utilizam uma conectividade sem fio (wireless). Portanto, torna-se cada vez mais imprescindível vincular a compressão de dados à proteção da ocorrência de erros desses dados compactados. Isto se deve principalmente por que no caso da transmissão de dados sem fio os erros são mais freqüentes e o tempo necessário para a recuperação é maior [HOF97].

A recuperação dos dados compactados no processo de descompactação pode se tornar muitas vezes inviável, no caso da ocorrência de erros durante a transmissão dos dados. Tomando-se como exemplo o código de compressão LZW, o processo de descompactação poderia ser totalmente comprometido na ocorrência de erros no dicionário, pois este, por corresponder à parte mais importante do código, poderia provocar a propagação do erro, impossibilitando a reconstrução dos dados. Em função da existência de fatores que poderiam acarretar a degradação ou total perda dos dados transmitidos, surge a necessidade de se empregar técnicas de Proteção Desigual de Erros (UEP – Unequal Error Protection).

Para as transmissões em tempo real, a utilização de técnicas convencionais para o controle de erro e a recuperação dos dados não é efetiva. O emprego do reenvio de pacotes de dados para a recuperação de erro de um canal de transmissão, não corresponde a uma forma prática de tratamento dos dados compactados de áudio e vídeo. Os atrasos ocasionados pela retransmissão interferem na execução de aplicações que operam em tempo-real. Neste caso,

uma estratégia muito mais efetiva seria limitar os efeitos do erro de canal para uma pequena parte dos dados transmitidos [HOF97].

Um dos esquemas empregados para a proteção desigual envolve a quebra dos dados compactados em blocos e a cada um desses blocos são inseridos cabeçalhos. Tais cabeçalhos possuem informações que descrevem os dados não compactados limitando a propagação de erro [HOF97]. Um outro esquema adotado para a detecção ou até mesmo a correção de erro compreende a inserção de mais redundância em cada bloco. Esta técnica requer a adição de um campo no início, no final ou então disperso dentro de cada bloco. Isto possibilitará a detecção de erro através da Verificação por Redundância Cíclica (CRC - Cyclic Redundancy Check) ou da correção de erro através de um Código Corretor de Erros (ECC – Error Correction Code) específico ao bloco [HOF97].

A compressão dos dados exerce um papel muito importante na transmissão e no armazenamento de informações. Cabe salientar que muitas vezes a normatização dos algoritmos de compressão, visando a interoperabilidade entre sistemas, é imposta a uma determinada aplicação. Esta imposição torna-se evidente entre as aplicações onde a largura de banda é insuficiente ou extremamente cara. Além disso, existindo a necessidade de se empregar a compactação dos dados num determinado sistema, serão exigidos maior tempo de processamento e maior complexidade, no que se refere a recuperação dos dados compactados.

1.3. Proposta

Uma vez que a ocorrência de erros nos dados compactados compromete a descompactação, principalmente com uma aplicação voltada à transmissão sem fio, torna-se relevante o estudo dos códigos de compressão usando proteção desigual de erros. Neste caso, a parte relevante dos dados compactados estaria sendo melhor protegida do que a parte secundária do código de compressão adotado.

A proposta deste trabalho está dividida em duas etapas distintas. A primeira etapa visa o estudo do impacto de erros no algoritmo de compressão LZSS, avaliando a parte da informação compactada mais sensível à propagação de erros. A segunda etapa consiste em propor uma estratégia de proteção desigual de erros, de maneira a minimizar a propagação de erros no processo de descompressão.

1.4. Organização

Este documento está organizado da forma a seguir. O Capítulo 2 apresenta os algoritmos de compressão baseados em dicionário, LZ77, LZ78, LZW e LZSS, e apresenta uma comparação entre LZW e o LZSS. O Capítulo 3 aprofunda o estudo do algoritmo LZSS, escolhido para este estudo. O Capítulo 4 descreve os conceitos básicos sobre proteção desigual de erros (UEP - Unequal Error Protection), modelos de canais, os conceitos de grupos e campos, e o código corretor de erro Reed-Solomon. O Capítulo 5 apresenta os resultados obtidos nos testes utilizando o algoritmo LZSS, o código Reed-Solomon e os gráficos correspondentes do impacto da propagação dos erros no arquivo de teste. As conclusões acerca dos resultados obtidos são apresentadas no Capítulo 6.

1.5. Contribuições

Este estudo estabelece um comparativo teórico entre os códigos de compressão LZ77 e LZSS, apresentando as vantagens deste último código em relação ao primeiro. A constituição dos tokens de ambos os códigos e as modificações implementadas no código LZSS que favorece o seu desempenho em relação ao seu precursor, são apresentadas na subseção 3.2.4.

Para a análise da estrutura dos arquivos compactados, foi necessário o desenvolvimento de um software utilizando-se do aplicativo MATLAB®. O software permite identificar os diferentes efeitos da propagação do erro quando da inserção de surtos de erro sobre qualquer um dos componentes que constituem o arquivo compactado.

Para a identificação do impacto da propagação dos erros após a inserção isolada de surtos de erro em cada um dos componentes do arquivo compactado, um método de comparação foi empregado. Através da comparação entre o arquivo original e o mesmo arquivo descompactado resultante da inserção de surtos de erro, verificaram-se as classes de informação mais sensíveis à ocorrência de erros.

Por meio da análise dos diferentes resultados, propôs-se um esquema de proteção desigual de erros para o algoritmo de compressão LZSS que poderá ser visto no capítulo 5.

Capítulo 2

Algoritmos de Compressão Baseados em Dicionário

2.1 Introdução

Quando se trata de algoritmos de compressão, está se referindo, na verdade, a dois algoritmos, os algoritmos de compressão e os algoritmos de descompressão. Os algoritmos de compressão que utilizam uma entrada *X* e geram uma representação *Xc* que requerem uma menor quantidade de bits, e os algoritmos de reconstrução que operam na representação compactada *Xc* para gerar a reconstrução *Y*. Estas operações são representadas na Figura 2.1. [HOF97]

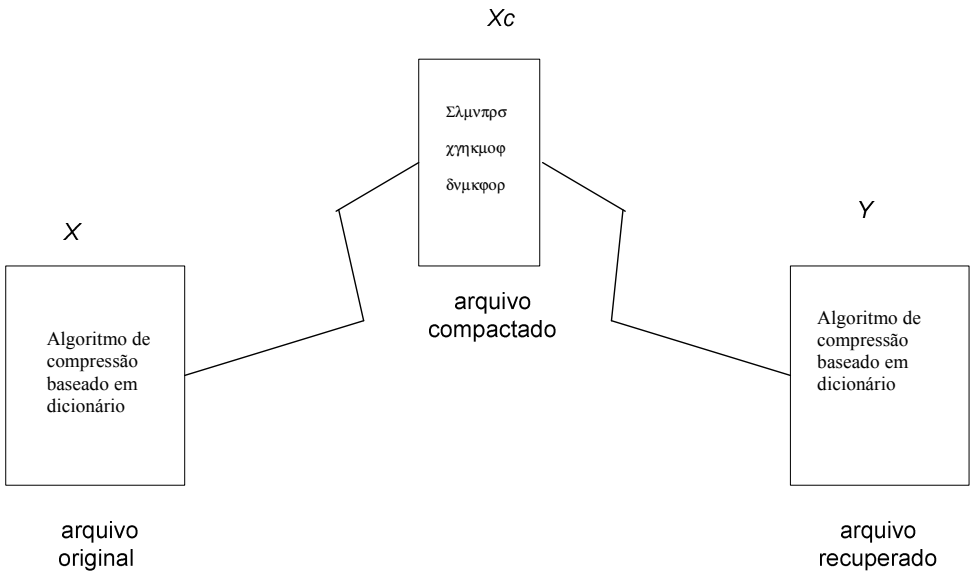

Fig. 2.1. Compressão e descompressão. [HOF97]

Baseados nos requisitos da compressão, esquemas de compressão de dados podem ser divididos em duas classes: esquemas de compressão sem perda (lossless compression), no qual *Y* é idêntico a *X*, e compressão com perdas (lossy compression), o qual normalmente apresenta maior taxa de compressão do que o esquema de compressão sem perdas, no entanto, permite que *Y* seja diferente de *X*.

A necessidade de se representar a informação eficientemente, particularmente texto, está presente na humanidade desde que o homem aprendeu a escrever. Abreviações, siglas ou símbolos são usados para representar letras do alfabeto ou ainda palavras e frases. Isto é uma forma de compactação para a escrita. Com a invenção dos equipamentos de gravação de áudio, a necessidade de se escrever tão rápido quanto se fala praticamente desapareceu. [HOF97]

A evolução da compressão de dados é apresentada na Figura 2.2.

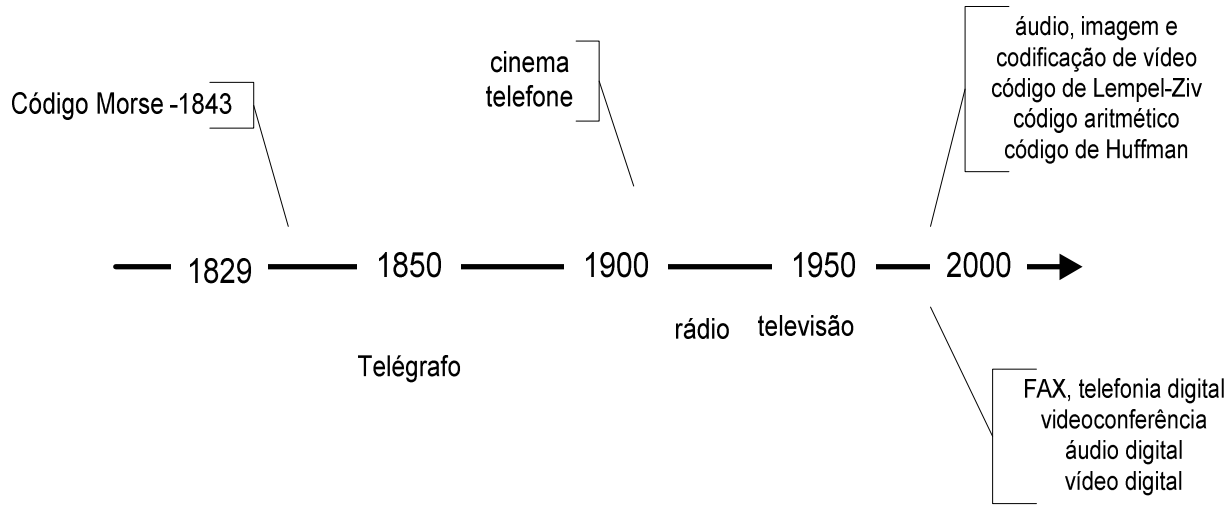

Fig.2.2. Linha do tempo da compressão de dados [HOF97]

2.2 Compressão sem perdas

Para a técnica de compressão sem perdas, como o próprio nome indica, não ocorre a perda de informação. Se os dados foram compactados sem perda, os dados originais podem ser recuperados exatamente dos dados compactados. A compressão sem perdas é normalmente usada para aplicações que não permitem qualquer diferença entre o dado original e o reconstruído.

A compressão de texto é uma área de grande aplicabilidade da compressão sem perdas. É muito importante que a reconstrução seja idêntica ao texto original, a menor diferença pode resultar em sentenças com significados totalmente distorcidos. Considerando-se o efeito que pode ser causado a uma frase, pode-se aplicar o mesmo argumento aos arquivos de computadores e a certos tipos de dados tais como registros de banco.

É importante que a integridade do dado, seja ele de que tipo for, seja preservada para o caso em que este tenha que ser processado ou incrementado com mais informação.

Por exemplo, supondo que uma imagem radiológica foi compactada seguindo o esquema de compressão com perdas e que a diferença entre a reconstrução Y e o original X era visualmente imperceptível. Se a esta imagem forem adicionados em outro instante mais detalhes, as diferenças anteriormente não detectadas podem causar o aparecimento de alguns falsos detalhes que podem confundir seriamente o diagnóstico do radiologista. Como o preço para este tipo de falha pode ser a vida humana, faz sentido o cuidado no que se refere a utilização do esquema de compressão que gera uma reconstrução que seja diferente do original.

Os dados obtidos dos satélites são freqüentemente processados posteriormente para se obter indicadores numéricos diferentes de vegetação, desmatamento, e assim por diante. Se os dados reconstruídos não são idênticos aos dados originais, o processamento pode resultar no incremento das diferenças. Pode não ser possível retornar e obter o mesmo dado. Portanto, não é aconselhável permitir qualquer diferença no processo de compressão.

2.2.1. Técnicas usadas na compressão sem perdas

De maneira geral, podemos classificar os algoritmos de compressão sem perdas mais utilizados como:

- Run Length Coding – um esquema simples e rápido que substitui os padrões repetidos por padrões e seus números de repetições.
- Códigos de Hufmann – utilizado para transmissões assíncronas.
- Códigos de Lempel-Ziv – utilizado para transmissões síncronas. [WON01]

A compressão sem perdas de dados simbólicos é alcançada mediante a criação de palavras-código que são menores que os símbolos correspondentes que codificam, se estes símbolos são comuns. Na Tabela 2.1, observam-se os tipos de codificação de dados simbólicos.

Tabela 2.1 Codificação de dados

Método de Codificação	Entrada	Saída
Fixo para variável	Um símbolo	Número variável de bits
Variável para fixo	String de símbolos	Número fixo de bits (bytes)
Variável para variável	String de símbolos	Número variável de bits

Fonte: [HOF97]

Historicamente, os primeiros métodos foram os de comprimento fixo para variável. Os algoritmos de Lempel-Ziv baseados em dicionário e a codificação Run-length são exemplos de codificação variável para fixo. Os códigos aritméticos e os códigos de Huffman são exemplos de codificação fixa para variável. [HOF97]

Existem muitas situações que se deseja um nível de compressão que possibilite uma reconstrução idêntica ao original. Por outro lado, há inúmeras outras ocasiões nas quais é possível abrir mão desta precisão para se obter um maior grau de compressão. Nestas situações é que se encaixam as técnicas de compressão com perdas.

2.3 Compressão com perdas

A compressão com perdas envolve alguma perda de informação, e os dados que tenham sido compactados utilizando-se da compressão com perdas geralmente não poderão ser recuperados ou reconstruídos de forma exata. Em resposta a esta distorção na reconstrução, são obtidas maiores taxas de compressão do que seria possível na compressão sem perdas. Em muitas aplicações, esta não exatidão na reconstrução não é um problema. Por exemplo, quando armazenamos ou transmitimos voz, o valor exato de cada amostra da conversação não é necessário. Dependendo da qualidade requerida para a reconstrução da conversação, quantidades variadas de perda de informação sobre o valor de cada amostra podem ser toleradas. Se a qualidade requerida para a reconstrução da conversação é similar àquela ouvida pelo telefone, uma perda significativa de qualidade pode ser tolerada. Entretanto, se a reconstrução da conversação necessita de uma qualidade comparada a um compact disc, a quantidade de perda de informação tolerada é relativamente baixa.

Similarmente, quando se vê a reconstrução de uma seqüência de vídeo, o fato de que a reconstrução é diferente da original, geralmente não é importante, enquanto as diferenças não

resultem em falsos detalhes. Por isto, em compressão de vídeo, geralmente é usado a compressão com perdas. Uma vez desenvolvido o esquema de compressão de dados, está-se apto a medir o seu desempenho. Devido às diferentes áreas de aplicação, termos diferentes têm sido desenvolvidos para a medição e a descrição do desempenho.

2.4. Algoritmo de compressão LZ77

No algoritmo de compressão LZ77 há uma seleção primária de uma seqüência de símbolos (string), a informação necessária para a compactação e descompactação é armazenada em um dicionário. Uma cópia desse dicionário é mantida em ambas as fases (compactação e descompactação) do sistema. Este tipo de abordagem torna-se extremamente efetivo no caso da compressão de textos onde uma seqüência de caracteres representando palavras ocorre freqüentemente [HOF97].

Enquanto a codificação estatística substitui cada símbolo por uma palavra código, a codificação por dicionário substitui um conjunto de símbolos por um único token (tripla) composto por três itens [HEL96]:

- Offset (o_i) – compreende a distância do ponteiro de referência do look-ahead buffer
- Match length ou Comprimento da string (f_i)– compreende o número de símbolos consecutivos no search buffer (dicionário) que são idênticos aos símbolos consecutivos no look-ahead buffer, iniciando com o primeiro símbolo referenciado pelo ponteiro.
- Primeiro símbolo do look-ahead buffer subseqüente à frase (u_i)

O token utilizará uma quantidade menor de bits do que a palavra a ser codificada, justificando-se desta forma a redução do tamanho do arquivo.

A Figura 2.3 ilustra um diagrama em blocos do fluxo de dados do código LZ77 onde a repetição de cada string de símbolos é substituída por um token apontando para uma ocorrência daquela string [HOF97]. Inicialmente, tanto o search buffer como o look-ahead buffer encontram-se vazios e representam uma janela móvel com *n* caracteres. O fluxo de dados passa primeiramente pelo look-ahead buffer e em seguida pelo dicionário que representa a parte codificada da janela deslizante [HEL96].

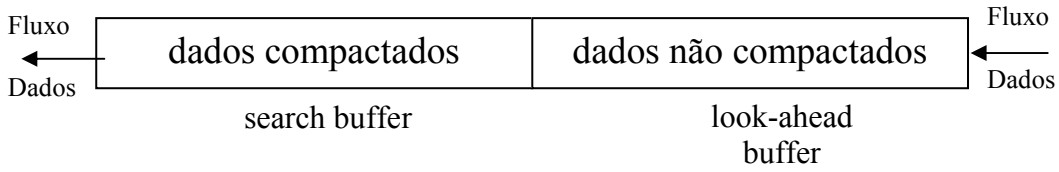

Fig. 2.3. Janela deslizante do LZ77 – Sliding Window [HOF97]

O dicionário contém bloco de dados codificados, enquanto o look-ahead buffer contém símbolos ainda não codificados. O tamanho de ambos os buffers é um parâmetro importante: se o dicionário for muito pequeno, a probabilidade de se encontrar a correspondência entre strings é muito menor. Se o dicionário for muito grande, a taxa de compressão será afetada, porque os ponteiros tornam-se grandes quando a string correspondente é encontrada. Além disso, o tempo de busca para se determinar a correspondência entre strings também aumenta [HOF97].

Além da dificuldade em se estabelecer um tamanho ideal entre o comprimento do dicionário e do look-ahead buffer, existe um outro problema que envolve a composição do próprio token que está relacionado com o caractere seguindo a string codificada. A utilização deste caractere torna-se desnecessária uma vez que este poderá ser incluído como parte do token subseqüente [SAY00].

De forma a ilustrar o processo de codificação LZ77, pode-se supor a seguinte seqüência a ser codificada:***....cabracadabrarrarrad...***

Conforme a Figura 2.4, o comprimento da janela é de 13 caracteres e o tamanho do look-ahead buffer possui 6 caracteres.

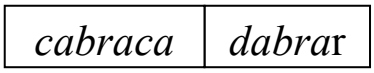

Fig. 2.4. Seqüência codificada [SAY00]

Com *dabrar* no look-ahead buffer busca-se na porção já codificada da janela uma correspondência para a letra *d.* Como pode-se observar, não há correspondência, portanto, transmite-se o token (0,0,C(d)). Os primeiros dois elementos do token indicam que não existe nenhuma correspondência a letra *d* no dicionário, enquanto que C(d) é o código para o caractere *d.* Esta técnica parece ser redundante quanto a forma de se codificar um único caractere [SAY00].

A Figura 2.5 ilustra o processo de codificação de uma string.

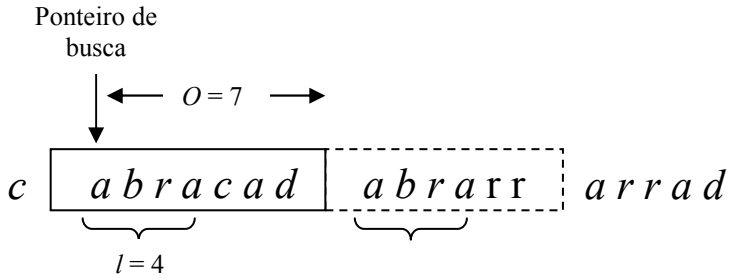

Fig. 2.5. O processo de codificação.[SAY00]

Como foi codificado um único caractere, a janela é deslocada de um caractere. Neste instante os conteúdos do search buffer (dicionário) e do look-ahead buffer se apresentam como indicado na Figura 2.6.

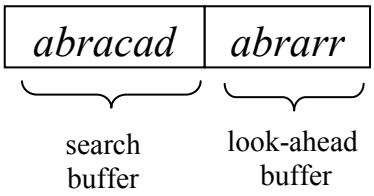

Fig. 2.6. Conteúdos após o deslocamento da janela [SAY00]

Considerando-se as posições atuais dos caracteres e analisando-se o dicionário da esquerda para a direita, observa-se uma correspondência a letra *a* no offset dois $(o = 2)$. O comprimento *l* desta correspondência é 1 $(l = 1)$. Deslocando-se um pouco mais para a esquerda no dicionário, tem-se uma nova correspondência para a letra *a* no offset quatro $(o = 4)$; mais uma vez, o comprimento *l* da correspondência é um $(l = 1)$. Após um novo deslocamento na janela, observa-se uma terceira correspondência para *a* num offset de sete $(o = 7)$. No entanto, para este caso o comprimento *l* é igual a quatro $(l = 4)$, conforme se observa na Figura 2.5. Portanto, a string *abra* é codificada com o token (7,4,C(r)), sendo que a janela é deslocada para a esquerda cinco caracteres [SAY00]. A Figura 2.7. ilustra a disposição dos caracteres no código de compressão.

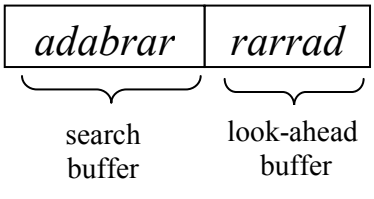

Fig. 2.7. Disposição dos caracteres no código de compressão [SAY00]

Para esta situação, analisando-se mais uma vez o dicionário da esquerda para a direita encontra-se uma correspondência para a letra *r* no offset um de comprimento também igual a um. Uma segunda correspondência se dá no offset três com um comprimento que a princípio também parece ser igual a três. No entanto, utiliza-se uma correspondência de comprimento igual a cinco ao invés de três [SAY00]. Este processo de decodificação funciona de acordo como é apresentado na Figura 2.8.

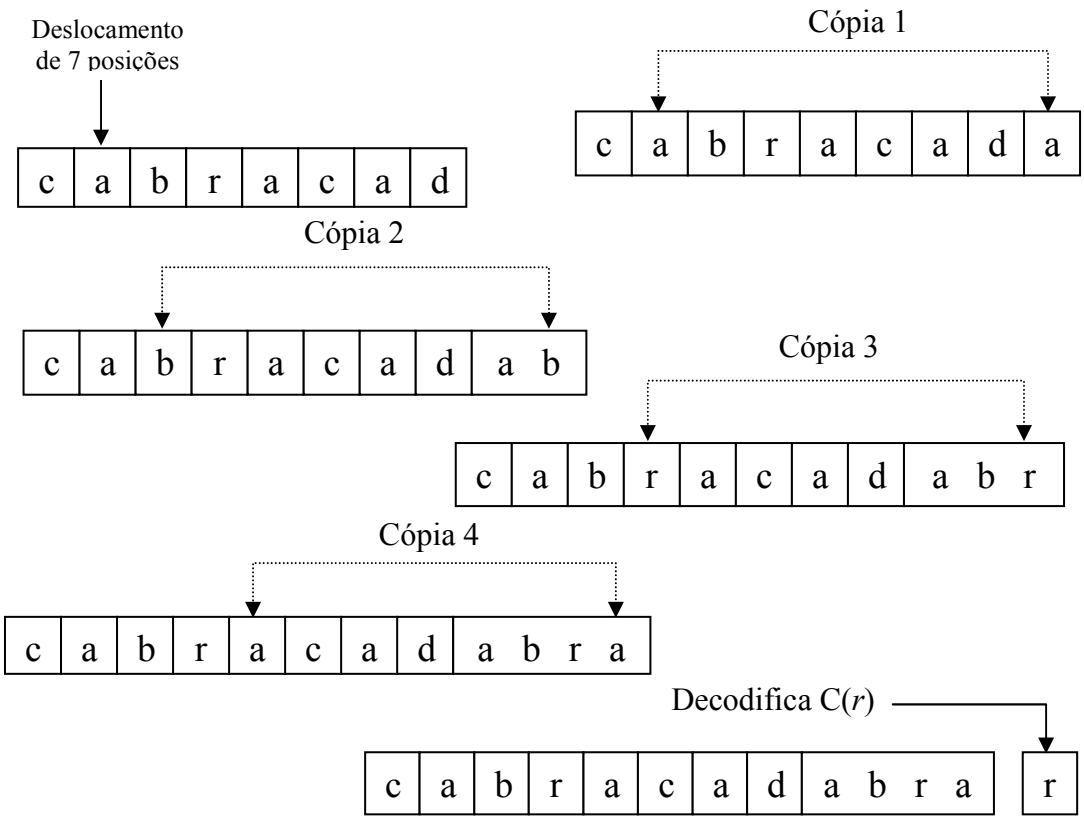

Fig. 2.8. Decodificação do token (7,4,C(r)).[SAY00]

Assumindo-se a codificação da seqüência *cabraca* e o recebimento dos tokens (0,0,C(d)), (7,4,C(r)) e (3,5,C(d)), torna-se relativamente simples a decodificação do primeiro token; não se tem nenhuma correspondência dentro da string previamente decodificada e o próximo símbolo é *d*. A string decodificada é agora *cabracad*. O primeiro elemento do token subseqüente informa ao decodificador que desloque o ponteiro de cópia sete caracteres à esquerda e copie quatro caracteres a partir daquele ponto.

Finalmente, para o token (3,5,C(d)) a decodificação se dá deslocando-se três caracteres para a esquerda iniciando-se a cópia. Os três primeiros caracteres a serem copiados são *rar*. O ponteiro de cópia move-se mais uma vez, como mostrado na Figura 2.9, para que se faça a cópia do caractere *r*. Da mesma maneira, copia-se o próximo caractere *a*. Mesmo iniciando-se a cópia de três caracteres, finalizou-se a decodificação de cinco caracteres [SAY00].

Observar que a correspondência tem apenas que se iniciar no dicionário, podendo se estender até o look-ahead buffer. Na verdade, se o último caractere no "lookeahead buffer" tivesse sido *r* ao invés de *d,* seguido de mais repetições de *rar,* a seqüência total de repetições de *rars* poderia ser codificada com um único token [SAY00].

Tabela 2.2. Resultado da Compressão

Seqüência.decodificada	Token recebido
Cabraca	...
Cabracad	(0,0,C(d))
cabracadabrar	(7,4,C(r))
cabracadabrarrarra	(3, 5, C(d))

Fonte: [SAY00]

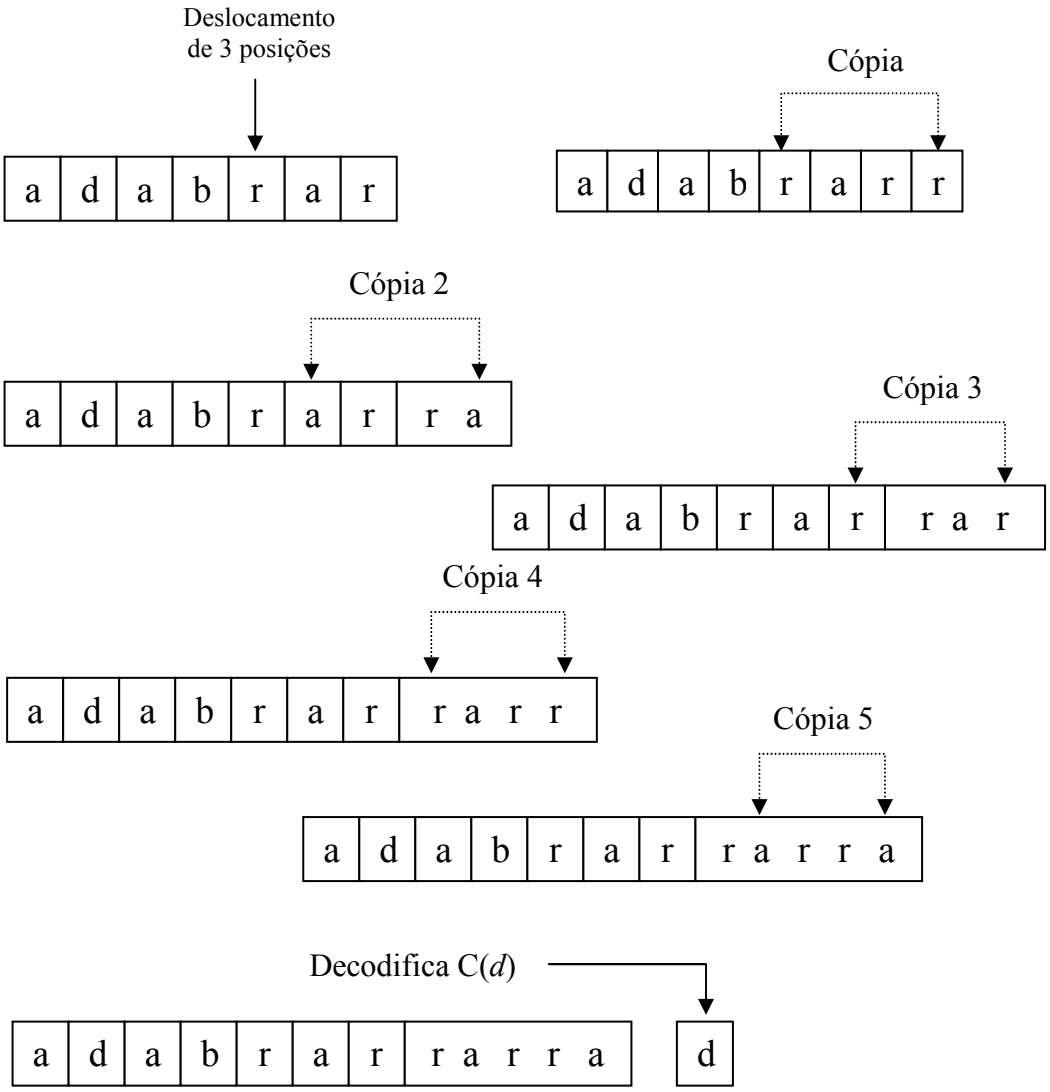

Figura 2.9. Decodificação do token (3,5,C(d)).[SAY00]

2.5. Algoritmo de compressão LZ78

O algoritmo de compressão LZ78 tem o seu nome derivado do ano de publicação do segundo trabalho dos autores Ziv e Lempel em 1978 [HEL96]. O desenvolvimento do algoritmo LZ78 superou uma deficiência chave do seu antecessor LZ77. Ocorria que as strings e caracteres do dicionário eram descartados a medida que novos dados eram processados. Portanto, se as frases mais comuns fossem mantidas, a eficiência resultante do algoritmo seria aumentada [HOF97].

O LZ78 é um algoritmo de maior simetria quando comparado ao LZ77, sendo que suas velocidades de compactação e descompactação são equivalentes. Quando comparado com o LZ77, a compressão pode ser maior porque as buscas ao buffer não são necessárias. A descompressão pode ser mais rápida porque o dicionário do LZ78 deve ser atualizado [HOF97].

Para a seqüência **"wabbaΦwaabaΦwabbaΦwabbaΦwooΦwooΦwoo",** tem-se que o símbolo Φ corresponde ao caractere de espaçamento.

Considerando-se que inicialmente o dicionário está vazio, os primeiros símbolos encontrados são codificados com o valor de índice igual a zero. As primeiras três saídas do codificador são (0,C(*w*)), (0,C(*a*)) e (0,C(*b*)), ficando o dicionário inicial parecido com a tabela 3 [SAY00].

O quarto símbolo corresponde à letra *b,* compreendendo a terceira entrada no dicionário. Se o próximo símbolo fosse incorporado, teríamos a seqüência *ba*. Como esta seqüência não se encontra no dicionário, os dois símbolos são codificados como (3,C(a)), sendo o padrão *ba* adicionado como a quarta entrada no dicionário. A saída do codificador e o dicionário se desenvolvem como ilustrado nas tabelas 2.3 e 2.4. Observar que as entradas no dicionário geralmente tendem a se tornar longas e caso uma sentença em particular se repita freqüentemente, depois de certo tempo, esta mesma sentença se torna uma entrada no dicionário [SAY00].

Apesar do algoritmo LZ78 ser capaz de capturar padrões e de mantê-los indefinidamente, tal característica pode, no entanto, gerar problemas mais sérios. Como pode ser observado no exemplo, o dicionário cresce de forma indefinida. Numa situação prática, o crescimento do dicionário deveria ser interrompido em um determinado estágio e a codificação ser tratada como um esquema de dicionário fixo [SAY00].

Tabela 2.3. Início do dicionário

Índice	Entrada
1	w
2	a
3	b

Fonte: [SAY00]

Tabela 2.4. Desenvolvimento do dicionário

Saída do Codificador	Índice	Entrada
(0,C(w))	1	w
(0,C(a))	2	a
(0,C(b))	3	b
(3,C(a))	4	ba
(0,C(Φ))	5	Φ
(1,C(a))	6	wa
(3,C(b))	7	bb
(2,C(Φ))	8	aΦ
(6,C(b))	9	wab
(4,C(Φ))	10	baΦ
(9,C(b))	11	wabb
(8,C(w))	12	aΦw
(0,C(o))	13	o
(13,C(Φ))	14	oΦ
(1,C(o))	15	wo
(14,C(w))	16	oΦw
(13,C(o))	17	oo

Fonte: [SAY00]

2.6. Algoritmo de compressão LZW

Entre os pesquisadores que fizeram alterações dos algoritmos básicos LZ77 e LZ78 que resultaram numa contribuição significante a operação da compressão de string baseada em dicionário, estava Terry Welch [HEL96]. Num artigo publicado na IEEE Computer, alguns problemas básicos associados ao algoritmo LZ78 foram eliminados [HEL96].

O algoritmo LZW considera inicialmente o conjunto de caracteres como 256 strings individuais de entrada de tabela cuja faixa de códigos varia de 0 a 255. O algoritmo operava sobre o próximo caractere na string de entrada dos caracteres como segue:

1. Se o caractere estiver na tabela, pegue o próximo caractere.
2. Se o caractere não estiver na tabela, retire o código da string por último conhecido e adicione a nova string a tabela.

Sobre o algoritmo LZW, os caracteres a partir da entrada da fonte de dados são lidos e utilizados de forma a produzir progressivamente strings cada vez maiores até que uma das strings não esteja no dicionário. Quando isso acontece, a última codificação da string conhecida é retirada e uma nova string é adicionada à tabela [HEL96]. O interessante sobre esta técnica é que o compactador e o descompactador sabem que a tabela de string inicial consiste de 256 caracteres no conjunto de caracteres. Uma vez que o algoritmo se utiliza de um código numérico, token, para indicar a posição de uma string na tabela de string do dicionário, há uma minimização do comprimento do token a medida que o dicionário começa a ser preenchido [HEL96]. Utilizando-se da mesma seqüência de entrada do código LZ78, “ **wabbaΦwaabaΦwabbaΦwabbaΦwooΦwooΦwoo** “, e assumindo que o alfabeto para esta fonte é {Φ, a, b, o, w}, o dicionário LZW se assemelha à tabela 2.5[SAY00].

Tabela 2.5. Dicionário LZW inicial

Índice	Entrada
1	Φ
2	a
3	b
4	o
5	w

Fonte: [SAY00]

O codificador primeiramente encontra a letra *w*. Este padrão está no dicionário, portanto, a próxima letra é concatenada ao primeiro caractere formando a seqüência *wa*. Como este padrão não se encontra no dicionário, codifica-se o caractere *w* com o índice de dicionário igual a 5, adiciona-se o padrão *wa* como sendo o sexto elemento do dicionário e inicia-se um novo padrão começando pela letra *a*. Como a letra *a* se encontra no dicionário, concatena-se o próximo elemento *b* para formar o padrão *ab*. Este padrão não se encontra no dicionário, então se codifica a letra *a* com valor de índice de dicionário igual a 2, adiciona-se o padrão *ab* ao dicionário como sendo o sétimo elemento e inicia-se a construção de um novo padrão com a letra *b*. O mesmo padrão com duas letras é mantido até que a letra *w* do segundo *wabba* seja alcançada. Até este ponto a saída do codificador se constitui apenas dos índices do dicionário inicial: 5 2 3 3 2 1. O dicionário se apresenta tal qual é ilustrado na tabela 2.6 [SAY00].

O próximo símbolo da seqüência é *a*. Concatenando a letra *a* com a letra *w*, tem-se o padrão *wa*. Este padrão já existe no dicionário, então lê-se o próximo símbolo *b*. Concatenando este último símbolo (*b*) com *wa*, tem-se o padrão *wab*. Como este padrão não existe no dicionário, ele será incluído em sua décima segunda entrada, iniciando-se em seguida um novo padrão com o símbolo b. A entrada wa também será codificada com valor de índice igual a 6.

Tabela 2.6. Construção da 12ª. entrada do dicionário LZW

Índice	Entrada
1	Φ
2	a
3	b
4	o
5	w
6	wa
7	ab
8	bb
9	ba
10	aΦ
11	Φb
12	w...

Fonte: [SAY00]

Observar que após uma série de entradas com duas letras, tem-se agora uma com três letras [SAY00]. O dicionário no final do processo de codificação é apresentado na tabela 2.7.

Para ilustrar o processo de decodificação do algoritmo LZW serão utilizadas as saídas do codificador do exemplo anterior, portanto, a seqüência anteriormente obtida foi **5 2 3 3 2 1 6 8 10 12 9 11 7 16 5 4 4 11 21 23 4.**

Esta mesma seqüência de saída do codificador compreende a seqüência de entrada do decodificador. O decodificador se utiliza do mesmo dicionário inicial que o codificador conforme demonstrado na tabela 2.8 [SAY00].

Tabela 2.7. Dicionário LZW para a codificação da seqüência wabbaΦwaabaΦwabbaΦwabbaΦwooΦwooΦwoo[2]

Índice	Entrada	Índice	Entrada
1	Φ	14	aΦw
2	A	15	wabb
3	B	16	baΦ
4	O	17	Φwa
5	W	18	abb
6	Wa	19	baΦw
7	Ab	20	wo
8	Bb	21	oo
9	Ba	22	aΦ
10	AΦ	23	Φwo
11	Φb	24	ooΦ
12	Wab	25	Φwoo
13	Bba		

Fonte: [SAY00]

O valor de índice 5 corresponde a letra *w*, portanto *w* é decodificado como o primeiro elemento da seqüência. Ao mesmo tempo, de forma a se igualar o procedimento de construção do codificador, inicia-se a construção do próximo elemento do dicionário que corresponde a letra *w*. Como este padrão já existe no dicionário, ele não será adicionado no dicionário dando-se então continuidade ao processo de decodificação. A próxima entrada do decodificador corresponde a letra *a.* Esta é decodificada e concatenada com o padrão atual para formar o padrão *wa*. Como este último não existe no dicionário, ele deverá ser adicionado como o sexto elemento e inicia-se um novo padrão começando com a letra *a*. As próximas quatro entradas 3 3 2 1 correspondem as letras *bbaΦ* e geram as entradas de dicionário *ab*, *bb*, *ba* e *aΦ*. O dicionário se assemelha a tabela 2.7 onde a 11[a.] entrada está em construção [SAY00].

Tabela 2.8. Construção da 11ª. entrada do dicionário LZW enquanto decodificando

Índice	Entrada
1	Φ
2	a
3	b
4	o
5	w
6	wa
7	ab
8	bb
9	ba
10	aΦ
11	Φ…

Fonte: [SAY00]

A entrada 6, que corresponde ao índice do padrão *wa*, é decodificada primeiramente a letra *w* e depois a letra *a.* A letra *w* é concatenada ao padrão existente que corresponde ao símbolo *Φ,* e forma o padrão *Φw.* Como *Φw* não existe no dicionário ele se torna a 11ª. entrada. O novo padrão se inicia com a letra *w*. Anteriormente foi decodificado a letra *a* que é concatenada a letra *w* obtendo-se o padrão *wa*. Este padrão está presente no dicionário sendo então decodificada a próxima entrada, 8, correspondendo a entrada *bb* no dicionário [SAY00]. O primeiro *b* é decodificado e concatenado ao padrão *wa* para se obter o padrão *wab* o qual não existe no dicionário. O padrão *wab* é adicionado como a 12ª. entrada no dicionário e um novo padrão com a letra b é iniciado. Este padrão por sua vez existe no dicionário, então o próximo elemento é decodificado na seqüência da saída do codificador. Seguindo este padrão, pode-se decodificar toda uma seqüência. Observar que o dicionário sendo construído pelo decodificador é idêntico ao construído pelo codificador [SAY00].

2.7. Algoritmo de compressão LZSS

Para o método de codificação LZSS, tanto o look-ahead buffer como o dicionário são usados da mesma maneira como no código LZ77, no entanto, a composição do token e a estrutura de dados do código LZSS foram modificadas. Tais alterações foram propostas por Storer e Szymanski em função dos problemas anteriormente descritos para o código LZ77 [HEL96]. Para o código LZ77, o codificador sempre manterá a sua estrutura (tokens) independentemente da composição do texto de entrada, seja ele uma seqüência de símbolos a ser compactada ou simplesmente o caractere original não compactado [HEL96].

Para o LZSS os tokens e os caracteres originais – plaintext - podem ser livremente alternados. Para que isso seja possível, deve-se preceder cada token e cada caractere original de um bit de prefixo. Um bit de prefixo no estado "1" indica um símbolo de 8 bits não codificado. Um bit de prefixo no estado "0" indica uma referência de dicionário constituída de um token. A primeira parte do token corresponde ao offset, enquanto que a segunda parte corresponde ao comprimento da string [HEL96].

A estrutura de dados utilizada no LZSS também sofreu algumas alterações. Embora o LZSS possua uma janela móvel assim como o código LZ77, as strings e mesmo os caracteres não codificados que passam pelo look-ahead buffer e depois pelo dicionário, são inseridas numa estrutura de árvore de dados. Utilizando-se de uma árvore binária para se fazer o armazenamento dos dados processados, tem-se um ganho no tempo de processamento em relação ao código LZ77. Comparativamente, duplicando-se o comprimento do dicionário para o código LZ77, duplica-se também o tempo de processamento [HEL96].

2.8. Conclusão

Neste capítulo foram apresentadas algumas características das técnicas de compressão com perdas e sem perdas. Como este estudo baseou-se em trabalhos que adotaram códigos de dicionário para a análise da eficiência da compressão, o algoritmo LZSS, descrito superficialmente neste capítulo e até então não utilizado para análise, foi empregado no estudo. O código LZSS, apresenta alguns aperfeiçoamentos em relação ao código LZ77 que foram detalhados no capítulo 3.

Capítulo 3

Estudo do Algoritmo LZSS

3.1. Introdução

Os algoritmos de compressão de dados transformam uma seqüência de símbolos em uma seqüência de palavras-código, dentro de um número finito de passos. Os algoritmos de Lempel-Ziv são exemplos de técnicas de codificação de dicionário de comprimento variável para fixo, onde *strings* (seqüência) de símbolos são substituídas por um único *token* que identifica uma entrada de dicionário contendo uma seqüência de símbolos. O codificador e o decodificador mantêm cópias idênticas do dicionário. Neste capítulo, aprofundaremos a descrição do algoritmo LZSS iniciada na Seção 2.5, do Capítulo 2.

3.2. Algoritmo de Compressão LZSS

Reconhecendo alguns problemas encontrados no LZ77, Storer e Szymanski [STO82] propuseram em 1982 uma modificação no algoritmo LZ77, que ficou conhecida por LZSS, em reconhecimento aos autores do artigo. No método de codificação LZSS, uma janela móvel de n caracteres com c caracteres formando um buffer de busca (lookahead buffer) e n-c caracteres formando o dicionário com seqüências (strings) previamente codificadas são usados da mesma forma que o algoritmo LZ77. Entretanto, a composição do ponteiro (token) e da estrutura de dados compactados foi modificada [HEL96].

Em seu artigo, Storer e Szymanski [STO82], detalharam as propriedades do modelo macro para compressão de dados, que passou a ser denominado posteriormente por LZSS. A seguinte notação foi por eles utilizada:

(1) Se s e s_l denotam *strings* e $n \geq 1$ é um inteiro, $s_1 s_2$ denota a concatenação de s_1 com s_2, $\prod_{l=1}^{n} s_l$ denota $s_1\ s_2\ ..\ s_n$ e s^n denota $\prod_{l=1}^{n} s_l$. s^0 denota a string vazia. Introduziu-se o termo **coleção** para significar conjunto variado. Um conjunto variado é um conjunto no qual repetições são permitidas. Por exemplo, {a,a,b} é um conjunto variado.

(2) Se s é uma *string*, $|s|$ denota o comprimento de s, e se s é uma **coleção**, $|s|$ denota o número de elementos em s (com cada elemento sendo contado quantas vezes aparece em s).

(3) A função *min* foi extendida para *strings* será definida por:

$$\min\{s_1,s_2\} = \begin{cases} s_1 & \text{se } |s_1| \geq |s_2| \\ s_2 & \text{caso contrário} \end{cases}$$

(4) Para um número real h, h denota o último inteiro maior ou igual h .

3.2.1 O modelo e as definições básicas

A fonte de dados é tratada como uma string finita sobre algum alfabeto. Com esquemas macro *externos*, uma fonte de strings é codificada como um par de strings, um dicionário e um esqueleto. O esqueleto contém caracteres do alfabeto de entrada, entrelaçada com ponteiros para os sub-strings do dicionário. O dicionário pode conter ponteiros para os sub-strings do dicionário. A fonte de strings é recuperada substituindo os strings de dicionário por ponteiros. Com esquemas macro internos, uma string é compactada por meio da substituição de ocorrências duplicadas de substrings, com ponteiros para outras ocorrências dos mesmos substrings. O resultado é uma única string de caracteres e ponteiros [STO82].

Na definição de Storer e Szymanski, $p \geq 1$ denota o tamanho do ponteiro, assumindo que todos os ponteiros possuem um tamanho uniforme. Se x é uma string contendo ponteiros, o comprimento de x, denotado por $|x|$, é definido como o número de caracteres em x mais p

vezes o número de ponteiros em *x*. Também consideram o ponteiro como um objeto indivisível, o qual em um modo não especificado, identifica sem ambigüidade alguma string para o qual é referido como o *alvo* de tal ponteiro. O modo como o ponteiro é escrito não é importante, a única consideração a ser feita é que sempre é possível determinar por inspeção de um ponteiro o comprimento de seu alvo. É possível escrever um ponteiro como um par (*n*,*m*) onde *n* indica a posição do primeiro caractere no alvo, *m* indica o comprimento do alvo, e |(*n*,*m*)| é o tamanho do ponteiro *p*. Pode-se assumir sempre que $m > p$ [STO82].

3.2.2 Exemplo de Aplicação

Seja $p = 1$, e considerando a string

$$w = aaBccDaacEaccFacac,$$

a qual deve ser codificada sobre o modelo externo macro como

$$x = aacc\#(1,2)B(3,2)D(1,3)E(2,3)F(2,2)(2,2),$$

onde # separa o dicionário do esqueleto. Por conveniência, assume-se que |#| = 0. A compressão atingida pela string *x* (isto é, a razão $|x|/|w|$) é 14/18. Usando o modelo macro interno, *w* pode ser codificado como

$$y = aaBccD(1,2)cEa(4,2)Fac(13,2),$$

atingindo uma compressão de 15/18.

Ciclos não podem ocorrer em formas comprimidas com ponteiros compactados, mas usando os ponteiros originais, ciclos podem freqüentemente fazer sentido. Por exemplo, a forma compactada *ab*(5,2)*a*(1,3) determina o palíndromo *abaaaaba* mesmo que os dois ponteiros na forma compactada formem um ciclo. Aqui os ponteiros (5,2) e (1,3) são um ciclo no sentido de que cada ponteiro aponta para uma porção da string representada pelo outro. Um exemplo de um ciclo degenerativo é dado pela forma compactada *a*(1,*n*), o qual determina o string a^{n+1}.

Storer e Szymanski [STO82] formularam quatro esquemas macro e três tipos de restrições, as quais podem ser aplicadas a qualquer um destes esquemas. A notação $\sum$ denota o alfabeto do qual os dados em questão são construídos.

Definição 1. Uma *forma compactada* de um string *S* usando o esquema EPM (external pointer macro) é qualquer string $t = S_0 \# S_1$ que satisfazem

(1) S_0 e S_1 consistem de caracteres de Σ e ponteiros para substrings de S_0

(2) *S* pode ser obtido de S_1 através dos seguintes passos:

(a) Substituição de cada ponteiro em S_1 com seu alvo.

(b) Repitição do passo a) até S_1 ser um ponteiro null.

Definição 2. Uma *forma compactada* de um string *S* usando o esquema CPM (compressed pointer macro) é qualquer string *t* que satisfaz

(1) *t* consiste de caracteres de Σ e ponteiros para substrings de *t*

(2) *S* pode ser obtido de *t* formando o string t#t e então decodificando como o esquema EPM.

Definição 3. Uma *forma compactada* de um string *S* usando o esquema OPM (original pointer macro) é qualquer string *t* que satisfaz

(1) *t* consiste de caracteres de Σ e ponteiros representando substrings de *t*

(2) *S* pode ser obtido de *t* por meio da reposição de cada poneiro (n,m) pela seqüência de ponteiros (n,1),(n+1,1)...,(n+m-1,1) e então decodificado como o esquema CPM, com a condição de que os ponteiros são considerados para ter comprimento 1.

Definição 4. Uma *forma compactada* de um string *S* usando o esquema OEPM (original external pointer macro) é qualquer string $t = S_0 \# S_1$ que satisfaz

(1) *t* consiste de caracteres de Σ e ponteiros.

(2) S_0 pode ser decodificado usando o esquema OPM para produzir um string *r*. Futuramente, ponteiros em S_1 apontam para subtrings de *r*.

(3) *S* pode ser obtido por meio da reposição de cada ponteiro em S_1 com seu alvo em *r*.

Definição 5. Um ponteiro de CPM (OPM) q_1 depende do ponteiro q_2 se o alvo de q_1 contém q_2 (todo ou parte do string representada por q_2) ou se há um ponteiro q_3 tal que q_1 depende de q_3 e q_3 depende de q_2. Um esquema macro é limitado para *no recursion s*e ponteiros dependentes são proibidos, e para *topological recursion* se nenhum ponteiro depende dele próprio. [STO82]

Definição 6. Dois ponteiros se superpõem se os seus alvos se superpõem e estritamente se superpõem se seus alvos se superpõem mas nenhum alvo é um substring do outro. Um esquema macro é restrito para sem superposição se ponteiros superpostos são proibidos.

Definição 7. Um ponteiro de CPM (OPM) q aponta para a esquerda se o caractere mais à esquerda de seu alvo está para a esquerda de q (o caractere mais à esquerda do string representado por q). Um ponteiro à direita é similarmente definido. Um esquema macro é restrito para *ponteiros unidirecionais* se todos os ponteiros devem apontar para a mesma direção (certamente, com os esquemas EPM e OEPM, isto somente se aplica ao dicionário externo). Como um caso especial, é possível restringir um esquema macro para ter somente ponteiros à esquerda ou à direita [STO82].

As diferentes combinações dos quatro esquemas macro básicos e o *recursion*, superposição e direção de ponteiro, provêem um grande número de métodos de compressão de dados. As combinações são suficientemente genéricas para cobrir virtualmente todos os esquemas de substituição propostos na literatura. [STO82]

3.2.3 O Modelo Macro Externo

O modelo macro externo considera a coleção de macros como se estivesse fora do restante do string compactado. Isto torna os esquemas externos ideais para comprimir coleção de strings usando um **dicionário comum**. Há diversas razões porque é mais natural tratar todos os ponteiros como ponteiros compactados quando se discute este modelo. Primeiro, autores no passado têm usado o esquema EPM e não o esquema OEPM. Segundo, é permitido

descompactar porções arbitrárias de dados sem ter que produzir o string inteiro. Terceiro, os ponteiros compactados freqüentemente requerem menos espaço que os ponteiros originais. Contudo, há vantagens em se utilizar ponteiros originais em relação aos ponteiros compactados, o que justifica considerar o modelo OEPM [STO82].

Esquemas macro são baseados no princípio de encontrar strings redundantes e substituí-los por ponteiros. Diferentes variações de esquemas macro podem ser definidas especificando o significado de um ponteiro, isto é, um ponteiro pode indicar um substring de um string compactado, um substring do string original, ou um substring de algum outro string como um **dicionário externo**.

As maneiras de determinar o tamanho do ponteiro são tanto requerer que o conteúdo da informação de um ponteiro seja suficiente para distinguir todos os ponteiros em uma codificação, como requerer que um ponteiro esteja apto para identificar qualquer substring do dicionário. Para este fim, se t é uma codificação de algum string usando o esquema EPM, seja $\delta(t)$ o número de ponteiros distintos em t, e seja $d(t)$ a **porção de dicionário** de t [STO82].

3.2.4 Modificações do LZSS em relação ao LZ77

O método de codificação LZ77 se utiliza de uma janela deslizante de n caracteres, com c caracteres formando o *lookahead buffer* e n-c caracteres formando o dicionário das seqüências previamente codificadas. No entanto, a composição da tripla (token) e da estrutura de dados foi modificada [HEL96].

A tripla no LZ77 corresponde a uma palavra código composta de três itens (o_i, f_i, u_i), onde o_i compreende a posição da correspondência do primeiro caractere do *lookahead buffer* no dicionário, f_i estabelece o comprimento da mesma seqüência de caracteres existente no *lookahead buffer* e no dicionário e u_i compreende o primeiro símbolo do *lookahead buffer* desigual à seqüência de caracteres [FUJ00]. Enquanto a codificação LZ77 necessita alternar os ponteiros (o_i, f_i, u_i) com os caracteres não codificados (plain text) independentemente da composição do texto de entrada, na codificação LZSS as triplas e os caracteres não codificados podem ser recombinados livremente [FUJ00], assim como pode ser observado na Figura 3.1. Um único bit de *flag* precede cada tripla ou caractere não codificado. O bit de flag com nível lógico ‘1’ indica que um símbolo não codificado de 8 bits está por vir. O bit de flag

com nível lógico ‘0’ indica que uma referência de dicionário composta de duas partes está por vir. O terceiro componente da tripla, u_i, presente no código LZ77, é considerado desnecessário para o código LZSS, uma vez que pode ser freqüentemente incluído na tripla subseqüente [HEL96].

No que se refere ao uso de *tokens*, sobre o algoritmo LZ77, a tripla é constituída de offset, comprimento de string e o caractere seguinte da string [HEL96].

A primeira parte do token é o offset, e a segunda é o comprimento da string. A Figura 3.1 ilustra o formato da saída de dados do LZSS.

texto não codificado	<1><símbolo de 8 bits>
referência do dicionário	<0><posição><comprimento da seqüência>

Fig.3.1 – Formato dos dados compactados do código LZSS [HEL96]

Examinando o formato da referência de dicionário ilustrado na Figura 3.1, embora o código LZSS continue a armazenar os dados em uma janela de deslocamento que percorre cada substring ou caractere não codificado correspondendo ao texto a ser compactado, este ainda insere os dados em uma estrutura de árvore binária. Em comparação a estrutura de dados do LZ77, o LZSS ao utilizar uma árvore binária para o armazenamento dos dados processados, reduz o tempo necessário para a localização da maior correspondência entre os mesmos [HEL96].

A influência dos erros ocorridos durante a formatação dos componentes da seqüência de dados é diferente. Se o erro está no comprimento da seqüência, o comprimento da frase copiada é alterado, afetando a janela de deslocamento [FUJ00].

Conforme visto no Capítulo 2, cada token (o_i., f_i, u_i) do código LZ77, contém um caractere explícito, u_i, que assegura a operação apropriada quando a seqüência de caracteres não casa com nenhuma posição na janela de deslocamento, mesmo assim, este mesmo caractere pode ser codificado como parte do próxima token.

O algoritmo LZSS soluciona este problema usando uma combinação de ponteiros e caracteres, sendo estes últimos incluídos sempre que um ponteiro ocupar mais espaço que os próprios caracteres codificados [YUA96].

O algoritmo LZSS possui um parâmetro p que corresponde ao comprimento mínimo de strings correspondentes. O algoritmo LZSS compreende:

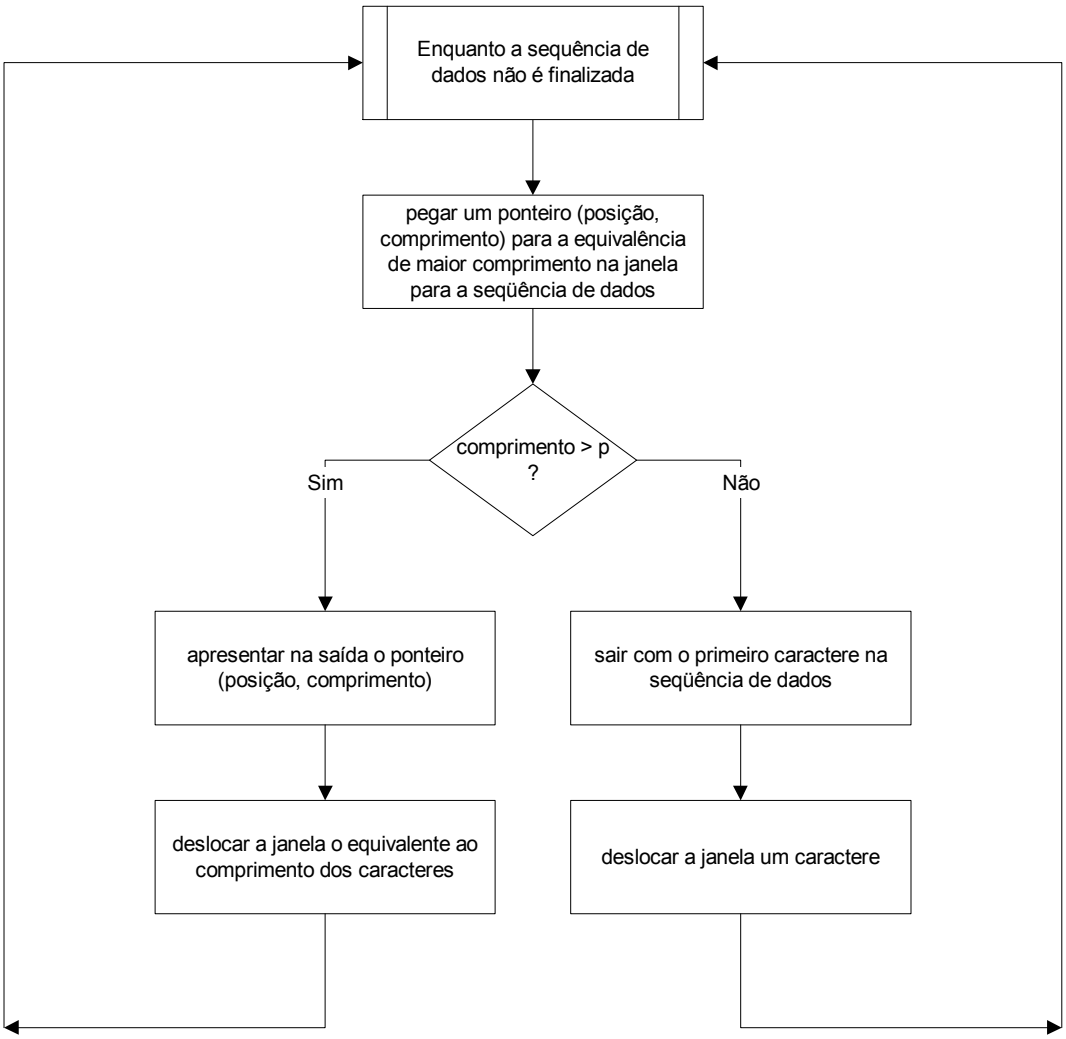

Fig.3.2 – Fluxograma do algoritmo LZSS [YUA96]

O formato de saída dos ponteiros é 1xx...xyy...y onde 1 é o *flag* de saída do ponteiro, xx...x é mapa de bits de o_i, yy...y é o mapa de bits de f_i, os números dos bits o_i e f_i são definidos pelo tamanho da janela e pelo comprimento do match string. A saída de caracteres é 0xxxxxxxx, e é constituído de um bit de flag e oito bits do caractere ASCII. [YUA96].

3.2.5 Vantagem do Algoritmo LZSS em Relação ao LZ77

De acordo com as propriedades locais dos arquivos de dados, o match string ou o maior casamento de caracteres de uma seqüência de dados pode ser encontrado nas partes finais do texto já codificado. Então é possível usar o offset do casamento de caracteres e a seqüência de dados atual para substituir o o_i dos ponteiros que denota a posição na janela. O seu propósito é codificar o offset mais efetivamente do que a posição. Um método eficiente é separar o offset em dois grupos, grupo do offset curto e grupo do offset longo. Quando o offset corrente pertence ao grupo curto, pode-se usar menos bits para codificar este offset, então o comprimento médio do código é menor que no LZSS. O formato de saída é conforme segue (o tamanho da janela é 4096):

Caractere:	0	caractere de oito bits
String:	1	offset comprimento do match string
Offset curto:	1	oito bits quando offset < 256
Offset longo:	0	vinte bits quando 256 <offset <4096

Comparado ao algoritmo LZSS, quando se introduz o offset e usa-se código de comprimento variável do código para denotá-lo, a taxa de compressão é otimizada [YUA96].

3.3. Conclusão

Justificativa aqui é diferente da dada no capítulo 2

Este capítulo apresentou de forma detalhada, o algoritmo LZSS, desenvolvido por Storer e Szymanski, a partir do algoritmo LZ77.

No LZSS a composição do token, bem como a estrutura de dados, foram modificadas. Na composição do token, a utilização do terceiro elemento que corresponde ao caractere seguinte a seqüência codificada, não é mais necessária no LZSS, uma vez que pode ser incluído como parte do token subseqüente. Para o caso da estrutura de dados, houve uma redução do tempo para a localização da mais longa seqüência coincidente, entre a informação constante no lookahead buffer e a informação constante no dicionário.

Isto permite a duplicação do tamanho do dicionário. No entanto, se duplicarmos o tamanho do dicionário na estrutura de dados do código LZ77, teremos uma duplicação do tempo de processamento.

No Capítulo 4 será discutido sobre a descrição de proteção desigual de erros, estratégias de recuperação de erros para códigos de compressão LZW e LZ77, conceitos de grupos e campos, e o código de Reed-Solomon.

Capítulo 4

Proteção Desigual de Erros

4.1. Introdução

Em algumas aplicações de codificação de fontes analógicas, como compressão de voz e vídeo, a sensibilidade do decodificador aos erros nos símbolos codificados é tipicamente não uniforme. A qualidade do sinal analógico reconstruído é insensível a erros que afetam certas classes de símbolos enquanto que o sinal analógico se degrada abruptamente quando os erros afetam outras classes de símbolos. [CAI96]

Podemos assumir que o codificador de fonte produz quadros de símbolos binários. Os símbolos de cada quadro podem ser particionados em classes de diferentes importâncias ou sensibilidades.

No Capítulo 2 apresentamos as principais estratégias de compressão sem perdas baseadas em dicionário adaptativo. No Capítulo 3 descrevemos em detalhes o algoritmo de compressão LZSS que é o foco de nosso estudo. Os erros introduzidos no processo de armazenagem de arquivos compactados em mídias/memórias de alta capacidade ou na transmissão de arquivos compactados em redes *sem fio*, podem comprometer totalmente a integridade da informação no processo de descompressão.

Uma interrupção do código LZ77 do texto compactado ocorre quando uma porção da seqüência de bits compactados se perde. Como exemplo, um setor com falha no HD (Hard Disk) pode produzir uma interrupção em um arquivo compactado armazenado. O arquivo pode ser descompactado adequadamente até o início da interrupção, porém na descompactação as variáveis podem ser posicionadas apenas onde o texto foi perdido na interrupção. Quando o descompactador trabalha nos ponteiros de lookback seguindo a

interrupção, este identificará que muitos dos ponteiros direcionarão para a área de interrupção e portanto poderão efetuar cópias apenas de variáveis. Portanto, a área seguinte à interrupção terá texto legível derivada de caracteres e lookbacks dentro da área precedente à interrupção, juntamente com as variáveis derivadas de lookbacks dentro da área de interrupção. Como o descompactador processa o registro do texto compactado, os seus lookbacks continuarão a copiar combinações de texto e variáveis. [MOS98]

A ocorrência de apagamentos (perdas) ou erros na seqüência compactada também se aplica não apenas no contexto de armazenagem digital como na transmissão digital, em particular nos sistemas de transmissão sem fio, onde o canal é bastante suscetível a erros. Neste contexto a otimização conjunta de codificação fonte (compressão) e codificação de canal (detecção e correção de erros) podem trazer vantagens significativas do ponto de vista de recuperação parcial de dados em arquivos compactados corrompidos ou na melhoria da eficiência de transmissão e recuperação de erros em enlaces de rádio. [HOF97]

4.2. Modelos de Canais

Para o sistema típico de transmissão de dados, citado na Seção 1.2 do Capítulo 1, pode-se considerar que:

- se a saída do decodificador em um dado intervalo de tempodepende apenas do sinal transmitido neste mesmo intervalo de tempo, e não de qualquer transmissão prévia, o canal é dito sem memória;

- se a saída do decodificador em um dado intervalo depende do sinal transmitido nos intervalos anteriores, bem como o sinal transmitido no intervalo atual, o canal é considerado canal com memória. Um canal com variação no tempo da intensidade e/ou da fase relativa de qualquer ou todos os componentes de freqüência do sinal recebido devido à alterações nas características na via de propagação, é um bom exemplo de um canal com memória, visto que a transmissão por multipercusos (multipath) destrói a independência de um intervalo para outro intervalo. A representação apropriada de modelos para canais com memória é difícil , e a codificação para estes canais é normalmente feita isoladamente. [LIN83]

Em canais sem memória, o ruído afeta cada símbolo transmitido de forma independente. Considere como exemplo, o canal simétrico binário (Binary Simmetric Channel – BSC), cujo diagrama de transição é mostrado na Figura 4.1.

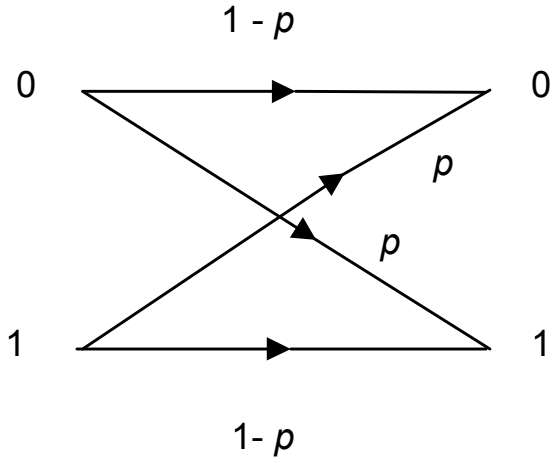

Fig 4.1 Canal Binário Simétrico (BSC). [LIN83]

Cada bit tem uma probabilidade *p* de ser recebido incorretamente e uma probabilidade *1-p* de ser recebido corretamente, independentemente dos outros bits transmitidos. Portanto os erros de transmissão ocorrem aleatoriamente na seqüência recebida, e canais *sem memória* são chamados de *canais com erros aleatórios*. Bons exemplos de canais com erros aleatórios são os canais de comunicação via satélite e comunicação espacial. A maioria das transmissões em linha de visada são primariamente afetadas por erros aleatórios.

Em canais com memória, o ruído não é independente de uma transmissão para outra. Um modelo simplificado de *canal com memória* é mostrado na Figura 4.2.

Este modelo contém dois estados, um "estado bom", no qual os erros de transmissão ocorrem de forma pouco freqüente, $p_1 \approx 0$, e um "estado ruim", no qual os erros de transmissão são altamente prováveis, $p_2 \approx 0{,}5$. O canal está no bom estado a maior parte do tempo, mas ocasionalmente desloca-se para o estado ruim, devido a uma mudança na característica de transmissão. Como conseqüência, os erros na transmissão ocorrem em surtos devido à alta probabilidade de transição no estado ruim. Os canais com memória são chamados de *canais com surtos de erros*. Exemplos de canais com surtos de erros são os canais de rádio, onde os surtos de erros são causados pelo desvanecimento do sinal devido ao multipercurso, e as transmissões por fio ou cabo, que são afetadas por ruído impulsivo.

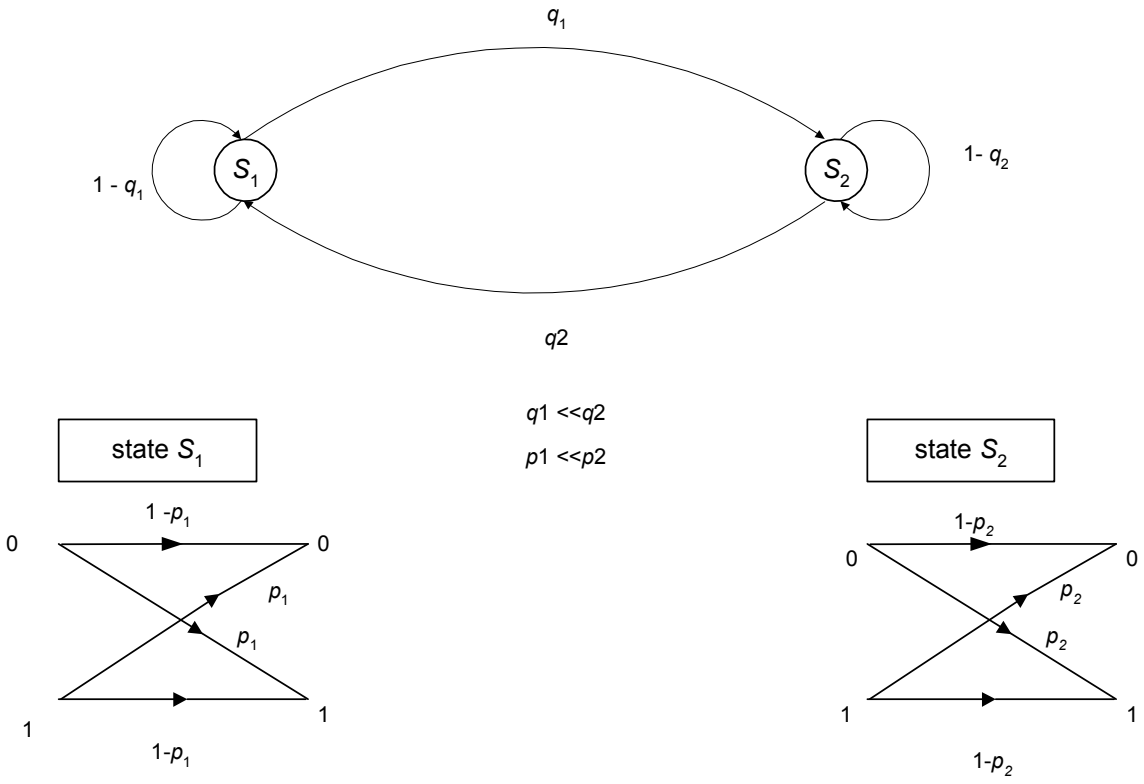

Fig. 4.2. Modelo de um canal com memória. [LIN83]

Outro exemplo é o processo de gravação magnética, que está sujeito a erros devido a defeitos na superfície da fita ou partículas de sujeira. Os códigos destinados a corrigir surtos de erro são chamados de códigos de correção de surtos de erro [LIN83]. Finalmente, existem canais que contém uma combinação de erros aleatórios e erros em surtos.

4.3. Estratégias de Controle de Erro

Em um sistema de comunicação unidirecional (simplex), a transmissão se faz estritamente do transmissor para o receptor. O controle de erros num sistema *simplex* é implementado aplicando-se códigos corretores de erro que corrigem automaticamente erros detectados no receptor. Esta estratégia é denominada *Forward Error Correction* (FEC). Podemos citar como exemplo os sistemas de armazenamento digital em fita magnética, na qual a informação gravada em fita pode ser reproduzida semanas ou meses depois de gravada [LIN83].

Nos casos em que a comunicação é bidirecional (duplex), a informação pode ser enviada em ambas as direções e o transmissor também age como um receptor (transceptor). O controle de erro de um sistema bidirecional pode ser implementado usando-se detecção de erro e um esquema de retransmissão, denominado de *Automatic Repeat Request* (ARQ). Em um sistema ARQ, quando os erros são detectados no receptor, uma requisição é enviada para o transmissor para repetir a mensagem. Este processo continua até que a mensagem seja recebida corretamente [LIN83]. A maior vantagem do ARQ sobre o FEC é que a detecção de erro requer um equipamento de decodificação mais simples que para correção de erros.

4.4. Estratégias de Proteção Desigual para Dados Compactados

Uma vez que os erros nos dados compactados provocam uma séria influência na descompactação, portanto o controle do erro para os dados compactados se faz necessária. A recuperação de erro utilizando-se dos códigos de LZW e LZ77, através de um esquema de proteção desigual de erro, possibilita uma proteção mais eficiente das partes mais importantes dos dados compactados. As partes ou componentes mais importantes dos dados compactados para o trabalho em questão correspondem às variáveis de bits de flag (flag), bits de offset (dicbits), bits de match (matchbits) e bits de caractere (caractere). A classificação de maior ou menor importância de tais componentes corresponde ao efeito da propagação dos erros sobre o arquivo descompactado quando inseridos surtos de erro em um desses componentes. A análise e os efeitos de propagação causados por cada um desses elementos podem ser visto no Capítulo 5, Conclusões e Recomendações. Nas subseções 4.4.1 e 4.4.2, estão detalhadas algumas propostas de recuperação de erro para os códigos LZW e LZ77.

4.4.1 Recuperação de Erros para o Código de Compressão LZW

Em [FUJ00] é proposto um esquema de proteção desigual para o algoritmo LZW. O algoritmo LZW foi descrito em detalhes na Seção 3.2 do Capítulo 3. No caso da recuperação de erros para o código LZW, considere que o alfabeto de entrada possua q caracteres. Inicialmente o dicionário contém q frases distintas, cada uma correspondendo a um caractere do alfabeto de entrada. Uma frase pode corresponder a uma palavra, a uma parte da palavra ou a várias palavras. O algoritmo de compressão procura no dicionário por uma frase que corresponda à entrada e armazena o ponteiro para a frase. Ao mesmo tempo, uma nova frase é

analisada e adicionada ao dicionário até que se atinja o limite do número de frases. Considere que o tamanho do dicionário seja de *M* frases. Portanto o comprimento do ponteiro é de $\lceil \log_2 M \rceil$ bits, onde $\lceil x \rceil$ corresponde ao menor inteiro maior ou igual a x, sendo que os primeiros $(M-q)$ ponteiros são usados para reconstruir o dicionário no processo de descompactação. Portanto, a primeira parte (search buffer) é mais importante e deve ser melhor protegida do que a segunda parte (lookahead buffer).

O esquema proposto em [FUJ00] para proteção desigual de erros segue o seguinte algoritmo:

1. Dividir os dados compactados em duas partes, onde a primeira parte dos dados (dicionário) consiste de $(M-q)\times(\log_2 M)$ bits e a parte secundária (lookahead buffer) consiste dos dados compactados restantes.

2. Aplicar à parte principal dos dados um código que corrige t_1 bytes de erros e um código que corrige t_2 bytes de erros da parte secundária dos dados, onde $t_1 > t_2$.

No esquema de proteção desigual de erros proposto e avaliado [FUJ00] adotou-se para teste um arquivo fonte com $M = 8192$ frases, $q = 256$ caracteres, $t_1 = 5$ e $t_2 = 1$. O arquivo fonte utilizado foi o "paper1" de [FUJ00]. O comprimento dos bits de verificação (redundância) foi de 116 bits e os surtos de erros inseridos no arquivo compactado foram de 48 bits. Para efeito de comparação, foi avaliado também o efeito de propagação de erros no arquivo compactado não protegido e no arquivo compactado protegido de maneira uniforme, aplicando-se um código convencional “4bEC” (four bytes error correction) com capacidade de correção de 4 bytes, com 116 bits de redundância [HEL96].

Os resultados de simulação obtidos em [FUJ00] mostraram que o esquema proposto com proteção desigual é mais eficiente no controle de erros do que o código convencional com proteção uniforme em todo o arquivo. O parâmetro de comparação foi o percentual de linhas erradas no arquivo descompactado em relação ao número total de linhas. O surto de erro de 48 bits foi inserido em diferentes posições ao longo do arquivo compactado.

4.4.2. Recuperação de Erros para o Código de Compressão LZ77

Em [FUJ00] também foi proposto um esquema de proteção desigual de erros para o código LZ77. O código LZ77 procura numa janela deslizante de tamanho fixo, pelas maiores frases que são iguais à seqüência de entrada atual. Na *i*-ésima correspondência, o arquivo de compressão gera um ponteiro de tamanho fixo (token) com os parâmetros (o_i, f_i, u_i). Estes parâmetros do ponteiro foram descritos em detalhes na subseção 3.2.4 do Capítulo 3. Basicamente o parâmetro o_i representa a posição inicial dentro do dicionário, f_i representa o tamanho da seqüência a ser copiada e u_i é o primeiro caractere distinto após a seqüência copiada. Na descompressão do *i*-ésimo ponteiro, o algoritmo de descompressão cópia a frase de f_i símbolos, cuja posição inicial no dicionário é indicada por o_i e desloca a frase copiada bem como o caractere u_i para dentro da janela deslizante e para o buffer de saída [FUJ00]. O efeito dos erros no componente f_i é bastante diferente dos efeitos em o_i ou u_i. Se o erro estiver nos parâmetros f_i, o comprimento da frase copiada será alterado. Isso afeta o deslocamento da janela, ou seja, os parâmetros o_i dos ponteiros subseqüentes vão indicar frases diferentes das frases corretas. As simulações apresentadas em [FUJ00] indicam que os erros ocorridos no componente f_i provocam em média 40 vezes mais danos do que se ocorridos nos componentes o_i ou u_i. Além disso, a ocorrência de erros nos f_i localizados na primeira parte dos dados compactados gera influências mais sérias do que os erros provocados na parte secundária. A seguir são descritas as etapas do algoritmo proposto em [FUJ00] para o LZ77:

a) Agrupar a saída compactada (o_1, f_1, u_1), (o_2, f_2, u_2), ... , (o_n, f_n, u_n) em duas seqüências, $\{o_1, u_1, o_2, u_2, \ldots, o_n, u_n\}$ e $\{f_1, f_2, \ldots, f_n\}$.

b) Aplicar um código corretor de erro, com capacidade de corrigir surtos de l_1 bits, à seqüência $\{o_1, u_1, o_2, u_2, \ldots, o_n, u_n\}$.

c) Aplicar um código corretor de erro com capacidade de corrigir surtos de l_2 bits, à seqüência $\{f_1, f_2, \ldots, f_{\lfloor n/2 \rfloor}\}$. Aplicar um código com capacidade de corrigir surtos de l_3 bits à seqüência $\{f_{\lfloor n/2 \rfloor+1}, f_{\lfloor n/2 \rfloor+2}, \ldots, f_n\}$, onde $l_2 > l_3$.

d) Adicionar os bits de redundância dos itens b) e c) no final dos dados compactados.

O esquema proposto foi implementado com $l_1 = 16$, $l_2 = 12$ e $l_3 = 8$. O comprimento da seqüência de redundância obtida é de 105 bits e o surto de erros inserido no arquivo é de 48 bits. O esquema proposto com proteção desigual de erros foi comparado com um esquema de codificação com proteção igual de erros, utilizando um código corretor, denominado *fire code*, com capacidade de corrigir surtos de erros de 40 bits. Este código adiciona 119 bits de redundância ao arquivo compactado. Os resultados obtidos em [FUJ00] demonstraram que o uso de proteção desigual traz vantagens significativas devido à natureza da estrutura do arquivo compactado.

4.5 Códigos para Proteção Desigual de Erros

Na transmissão e no processamento de dados, o nível de controle de erro desejado é assegurado através da utilização dos códigos corretores de erro. Em inúmeras importantes aplicações, no entanto, nem todos os dados são considerados de igual importância para o usuário. Portanto, é útil contar com códigos em que algumas informações são protegidas contra um número maior de erros do que outras informações. Tais códigos são chamados de códigos de proteção desigual de erro (do inglês: Unequal Error Protection - UEP). Boyarinov [BOY81] analisou em detalhes as propriedades dos códigos lineares de proteção desigual de erros sobre o campo de Galois GF(q).

Os primeiros a apresentarem tais códigos foram Masnick e Wolf [MAS67]. Estes descreveram algumas propriedades, encontraram relações e deram exemplos de códigos UEP lineares sistemáticos.

Quase todos os códigos algébricos previamente considerados na literatura, possuem a propriedade de que as suas capacidades de corrigir erros são descritas em termos de corrigir erros em palavras código preferencialmente do que corrigir erros em dígitos individuais em uma palavra código. Por exemplo, um código corretor com capacidade de correção de t-erros irá decodificar a palavra código corretamente se t ou menos erros ocorrerem na transmissão da palavra [MAS67].

Em seu artigo publicado em 1967, Masnick e Wolf [MAS67] descrevem que investigaram códigos nos quais certos dígitos de uma palavra código são protegidos contra um número maior de erros que os demais dígitos desta mesma palavra. Especificamente, cada

dígito da palavra código está em um nível de proteção de erro, denotado por f_i. Então, se f erros ocorrem na transmissão de uma palavra código, todos os dígitos para os quais $f_i \geq f$ serão decodificados corretamente, embora a palavra inteira possa ser decodificada incorretamente. Estes códigos são denominados *códigos de proteção desigual de erros*. Um exemplo onde códigos UEP podem encontrar aplicações é na transmissão de dígitos decimais codificados para binários. Considere a representação codificada para um inteiro *M* cuja magnitude pode variar entre 0 e $2^{\beta}-1$.

Portanto,

$$M = m_{\beta-1} \cdot 2^{\beta-1} + m_{\beta-2} \cdot 2^{\beta-2} + \ldots + m_0 \qquad \text{[MAS67]}$$

onde $m_i = 0$ ou $m_i = 1$. O inteiro *M* pode ser representado pelos coeficientes m_i como $m_{\beta-1} m_{\beta-2} \cdots m_0$. Se um erro modifica o coeficiente $m_{\beta-1}$, a magnitude de *M* pode ser modificada de $2^{\beta-1}$. Por outro lado, se um erro modifica o dígito m_0, a magnitude de *M* poderia ser modificada de um valor unitário. Desta forma, se erros de maior magnitude são mais importantes (dispendiosos) que os erros de menor magnitude, seria desejável encontrar um esquema de código que proporcione maior proteção para os dígitos de maior ordem do que para os dígitos de menor ordem [MAS67].

Em determinadas aplicações, poderia ser necessário se variar a quantidade de proteção de erro entre os diferentes blocos de dígitos da seqüência de informação. Considerando como exemplo o problema de um observador transmitindo simultaneamente resultados de diversos experimentos, alguns experimentos podem ser mais importantes do que outros, portanto, podem merecer maior proteção contra erros. Embora esta proteção desigual possa ser alcançada usando-se um código distinto para cada experimento, seria mais eficiente a utilização de um único código, e conseqüentemente um único codificador e um único decodificador. A UEP analisa o problema dos códigos lineares, para os quais a proteção designada para cada dígito ou conjunto de dígitos pode ser diferente da proteção de algum outro dígito ou conjunto de dígitos [MAS67].

Masnick apresenta alguns teoremas que justificam a utilização dos códigos UEP, pelo fato de que em certas situações diferentes pesos devem ser associados à diferentes dígitos da palavra código. O método clássico de avaliação da performance de um sistema de codificação pelo cálculo da probabilidade média de ocorrência de erro para uma palavra código não é

satisfatório para as situações que existem diferentes pesos para certos dígitos da palavra código, porque no método clássico todos os erros têm igual valor de importância. [MAS67]

Buschner introduziu o critério de erro numérico médio para as transmissões de dados numéricos quantizados.

Um valor v_j é designado para o *j*-ésimo dígito de informação de cada palavra código. Então a palavra código $\bar{C}_\alpha$ a qual contém os dígitos da informação $m_{\alpha,0}m_{\partial,1}\cdots m_{\alpha,k-1}$ é definida por $\sum_{j=0}^{k-1} v_j m_{\alpha,j}$. Se a palavra código $\bar{C}_\alpha$ é transmitida mas é decodificada como $\bar{C}_\beta$ no receptor, então o "custo" deste erro é definido como o valor absoluto da diferença dessas palavras código, isto é, $\left|\sum_{j=0}^{k-1} v_j m_{\alpha,j} - \sum_{j=0}^{k-1} v_j m_{\beta,j}\right|$. [MAS67]

Este critério transformou-se no termo discutido por Masnick como AEC - *Average Error Cost*. [MAS67]

4.6 Capacidade de Detecção e Correção de Erros dos Códigos de Bloco

Quando um vetor de código **v** é transmitido por canal com ruído, um padrão de erros de *l* erros irá resultar em um vetor recebido **r** o qual difere do vetor **v** transmitido em *l* posições [d (**v**,**r**) = l]. Se a distância mínima de um código de bloco C é d_{MIN} , qualquer dois vetores distintos de C diferem em pelo menos d_{MIN} posições. Para este código C qualquer padrão de erro de d_{MIN} - 1 ou menor quantidade de erros irá resultar em um vetor recebido **r** que não é uma palavra de código em C. Quando o receptor detecta que o vetor recebido não é uma palavra de código de C, dizemos que erros foram detectados. Um código de bloco com distância mínima d_{MIN} é capaz de detectar padrões de erros de d_{MIN}- 1 ou menor quantidade de erros.[LIN83]

4.7. Grupos e Campos

Os conceitos de grupos e campos são indispensáveis para a compreensão dos algoritmos de deteção e correção de erros [LIN83].

4.7.1 Grupos

Seja *G* um conjunto de elementos. Uma *operação binária* * em *G* é uma regra que assegura que para cada par de elementos *a* e *b*, um terceiro elemento único definido como *c*= *a***b* está em *G*. Quando uma operação binária * é definida em *G*, diz-se que *G* está *fechado* sob *. Por exemplo, seja *G* um conjunto de todos os inteiros e seja a operação binária em *G* a operação real de adição +. Para quaisquer dois números inteiros *i* e *j* em *G*, *i*+*j* é unicamente um inteiro definido em *G*. Então, o conjunto de inteiros está fechado sobre a operação de adição real. Uma operação binária * em *G* é dita associativa se, para qualquer *a*, *b* e *c* em *G*,

$$a*(b*c) = (a*b)*c \qquad \text{(eq. 4.1)}$$

Um conjunto *G* no qual uma operação binária é definida, é chamado um *grupo* se as seguintes condições são satisfeitas:

(i) A operação binária é associativa.

(ii) O conjunto *G* contém um elemento *e* tal que, para qualquer *a* em *G*, *a***e*=*e***a*=*a* . O elemento *e* é chamado o elemento identidade de *G*.

(iii) Para qualquer elemento *a* em *G*, existe outro elemento *a*' em *G* tal que *a***a*'=*a*'**a*=*e*. O elemento *a*' é chamado o inverso de *a* (*a* é também o inverso de *a*'). Um grupo *G* é chamado comutativo se sua operação binária * também satisfaz a seguinte condição: Para qualquer *a* e *b* em G, *a***b*=*b***a*.

4.7.2 Campos

O conceito de grupos auxilia a apresentar outro conceito algébrico, chamado *campo*. De forma simplificada, um campo é um conjunto de elementos nos quais é possível fazer adição, subtração, multiplicação, e divisão sem deixar o conjunto. Adição e multiplicação devem satisfazer as propriedades comutativa, associativa e distributiva [LIN83].

Definição: Seja *F* um conjunto de elementos no qual duas operações binárias, chamadas adição "+" e multiplicação " * " , são definidas. O conjunto *F* juntamente com as duas operações binárias "+" e " * " é um campo se as seguintes condições são satisfeitas:

(i) *F* é um grupo comutativo sobre a operação de adição +. O elemento identidade com respeito à adição é chamado o elemento *zero* ou a identidade aditiva de *F* e é denotada por *0*.

(ii) O conjunto de elementos diferentes de zero de *F* é um grupo comutativo sob multiplicação " * ". O elemento identidade com respeito à multiplicação é chamado o elemento unidade ou a identidade multiplicativa de *F* e é denotada por 1.

(iii) A operação de multiplicação é distributiva sobre a operação de adição, isto é, para quaisquer três elementos *a*, *b*, e *c*, em *F*, $a \cdot (b+c) = a \cdot b + a \cdot c$.

Isto vem da definição de que um campo consiste de pelo menos dois elementos, a identidade aditiva e a identidade multiplicativa. O número de elementos em um campo é chamado *ordem do campo*. Um campo com número finito de elementos é chamado campo finito [LIN83].

Exemplo: Considere o conjunto {0,1} com uma operação de adição e multiplicação módulo-2 conforme apresentado nas Tabelas 4.1 e 4.2. É possível confirmar que a operação de multiplicação módulo-2 é distributiva sobre a operação de adição módulo-2, simplesmente comutando $a \cdot (b+c)$ e $a \cdot b + a \cdot c$, para as oito combinações possíveis de *a*, *b* e *c*. (*a*=0 ou 1, *b*=0 ou 1, *c*=0 ou 1). Portanto, o conjunto {0,1} é um campo de dois elementos sob operação módulo-2 de adição e multiplicação.

Tabela 4.1 Módulo 2-adição

+	0	1
0	0	1
1	1	0

Fonte: [LIN83]

Tabela 4.2 Módulo 2-multiplicação

·	0	1
0	0	0
1	0	1

Fonte: [LIN83]

O campo do exemplo acima é usualmente chamado de campo binário e é denotado por *GF(2)*, ou *campo de Galois* de ordem 2. O campo binário *GF(2)* desempenha um importante papel na teoria de codificação e é largamente utilizado em computadores digitais e sistemas de transmissão ou armazenamento de dados digitais [LIN83].

Seja *p* um número primo. É possível demonstrar que o conjunto de inteiros {0,1,2,..., *p-1*}, é um grupo comutativo sobre a adição módulo-*p*. Da mesma forma, é possível demonstrar que os elementos diferentes de zero, {1,2,....., *p-1*} formam um grupo comutativo sobre a operação de multiplicação módulo-*p*. Portanto, o conjunto {0,1,2,....,*p-1*} é um campo de ordem *p* sob adição e multiplicação módulo *p*. Desde que o campo é construído a partir de um número primo *p*, é chamado de *campo primo* e é denotado por *GF(p)*. Para *p=2*, obtém-se o campo binário *GF(2)* [LIN83].

Para qualquer número primo *p*, existe um campo finito de *p* elementos. De fato, para qualquer inteiro positivo *m*, é possível estender o campo primo GF(*p*) para um campo de p^m elementos chamado de *campo de extensão* e é denotado por $GF(p^m)$. Campos finitos são também chamados *campos de Galois*. Uma grande porção da teoria algébrica de codificação, construção de códigos e decodificação é construído em torno de campos finitos. Desde que a aritmética dos campos finitos é muito similar à aritmética comum, muitas das regras da matemática comum se aplicam à aritmética dos campos finitos. Portanto, é possível utilizar muitas das técnicas de álgebra sobre campos finitos.

4.8 Código de Bloco Linear Reed-Solomon (RS)

Os códigos Reed-Solomon são códigos de bloco corretores de erro com uma vasta aplicação na comunicação digital e no armazenamento de dados. São utilizados para corrigir erros em muitos sistemas incluindo:

- Equipamentos de armazenamento (incluindo fita, CD, DVD, códigos de barra, etc)
- Comunicação móvel ou sem fio (incluindo aparelhos celulares, links de microondas, etc)
- Comunicação via satélite
- Televisão Digital
- Modems de alta velocidade tais como ADSL, xDSL, etc.

Um sistema típico é mostrado abaixo:

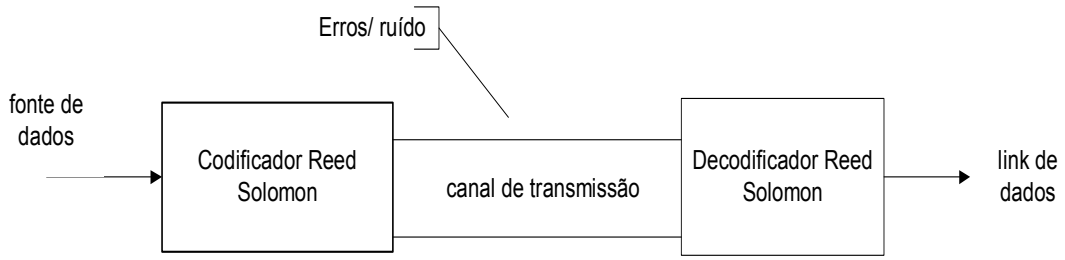

Fig.4.3 Sistema típico de codidificação RS.[COM04]

O decodificador Reed-Solomon insere bits "redundantes" extras em um bloco de dados. Durante a transmissão ou o armazenamento há a ocorrência de erros por inúmeras razões (por exemplo, ruído ou interferência, riscos em um CD, etc.). O decodificador Reed-Solomon processa cada bloco e procura corrigir erros e recuperar os dados originais. O número e o tipo de erros que podem ser corrigidos dependem das características do próprio código Reed-Solomon.

Os códigos Reed-Solomon são um subconjunto dos códigos BCH (Bose Chaudhuri Hoquingham) os quais são códigos de bloco lineares [LIN83]. Um código Reed-Solomon é especificado como RS (*n*,*k*) com *s*-bits. Isto significa que o codificador utiliza *k* símbolos com *s* bits cada um, e adiciona símbolos de paridade para fazer uma palavra de código com *n* símbolos. Existem *n*-*k* símbolos de paridade com *s* bits cada um. Cabe esclarecer que as denominações *n, k* e *s* são inerentes a definição do código Reed-Solomon conforme apresentado na literatura, não havendo para a variável *s* do código RS qualquer relação com a variável *s* definida no Capítulo 3, subseção 3.2. O código Reed-Solomon pode corrigir até *t* símbolos errados em uma palavra código, onde $2t = n–k$, conforme mostra a Fig. 4.4.

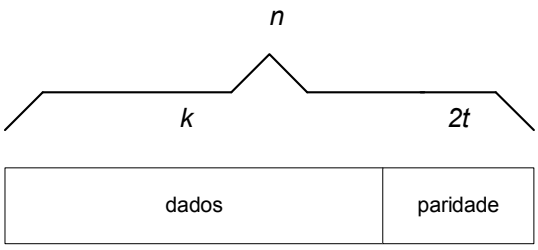

Fig.4.4 Sistema típico de codidificação RS.[COM04]

Exemplo: Para o código Reed-Solomon (255,223) com símbolos de 8 bits. Tem-se que cada palavra código possui um comprimento de 255 bytes dos quais 223 bytes são dados e 32 são bytes de paridade. Para este código:

$$n = 255, k = 223, s = 8$$
$$2t = 32, t = 16$$

O decodificador pode corrigir qualquer símbolo de 16 erros na palavra código: i.e: erros de até 16 bytes em qualquer lugar na palavra código podem ser corrigidos. [COM04]
Dado um símbolo de tamanho s, o comprimento máximo da palavra código (n) para o código Reed-Solomon é $n = 2^s - 1$. Por exemplo, o comprimento máximo de um código com 8 símbolos ($s = 8$) é 255 bytes.

Os códigos de Reed Solomon podem ser reduzidos (conceitualmente) fazendo um número de símbolos zero no codificador, não transmiti-los, e então reinserí-los no decodificador.

Exemplo: O código (255,223) descrito acima pode ser reduzido para (200,168). O codificador utiliza um bloco de 168 bytes de dados, (conceitualmente) adiciona 55 bytes zero, cria uma palavra código (255,223) e transmite apenas os 168 bytes de dados e 32 bytes de paridade.

A quantia de processamento necessária para codificar e decodificar os códigos Reed Solomon é relacionado ao número de símbolos de paridade por palavra código. Uma grande quantia de *t* significa que um grande número de erros podem ser corrigidos mas necessitam maior capacidade de processamento do que um pequeno valor de *t*.

Um símbolo de erro ocorre quando 1 bit em um símbolo está com erro ou quando todos os bits no símbolo estão errados.

Exemplo: RS (255,223) podem corrigir 16 símbolos com erro. No pior caso cada um dos 16 bits com erro podem ocorrer em um símbolo separado (byte), desta forma o decodificador corrige erros de 16 bit. No melhor caso, 16 bytes completos com erro ocorrem, sendo assim o decodificador corrige 16 x 8 erros de bit.

Os códigos Reed-Solomon são particularmente bem dimensionados para corrigirem surtos de erro.

Procedimentos da decodificação algébrica do código Reed-Solomon podem corrigir erros e eliminá-los. Uma eliminação ocorre quando a posição de um símbolo errado é conhecida. Um decodificador pode corrigir até t erros ou até $2t$ eliminações. A informação de eliminação pode ser freqüentemente fornecida pelo demodulador num sistema de comunicação digital; o demodulador indica os símbolos recebidos com maior probabilidade de conter erros.

Quando uma palavra código é decodificada, existem 3 possíveis resultados:

1. Se $2s + r < 2t$ (s erros, r eliminações) então o código original transmitido sempre será recuperado, por outro lado,

2. O decodificador detectará que este não pode recuperar a palavra código original e indica este fato, ou,

3. O decodificador recuperará uma palavra código indiretamente sem qualquer indicação.

A probabilidade de cada uma das três possibilidades depende do código Reed-Solomon em particular, do número e da distribuição de erros.

A vantagem de se utilizar o código Reed Solomon é que a probabilidade de um erro permanecer no dado decodificado é usualmente muito menor do que se o código Reed-Solomon não for usado. Isto é normalmente descrito como ganho de codificação.

Exemplo: Um sistema de comunicação digital é desenhado para operar com uma taxa de erro (BER) de10^{-9}, então significa que não mais que 1 em 10^9 bits são recebidos com erro. Isto pode ser alcançado pelo incremento da potência do transmissor ou pela inclusão do Reed-Solomon (ou outro tipo de FEC – Forward Error Correction). O Reed-Solomon permite ao sistema que atinja este BER com nível de potência de saída mais baixo. A potência não "dispendida" dada pelo Reed Solomon (em decibéis) corresponde ao ganho de codificação.

A codificação e a decodificação do código Reed Solomon podem ser implementadas em software ou em hardware para atender um propósito específico.

Os códigos de Reed-Solomon baseiam-se numa específica área da matemática denominada por Campos de Galois ou campos finitos. Um campo finito possui a propriedade de que as operações aritméticas (+, -, x, / etc.) nos elementos do campo sempre têm um resultado no campo. Um codificador ou decodificador Reed-Solomon necessita implementar estas operações aritméticas. Tais operações requerem funções de hardware ou software específicas para serem implementadas. [COM04]

Uma palavra código de Reed-Solomon é gerada utilizando-se de um polinômio especial. Todas as palavras código válidas são divisíveis pelo gerador polinomial. A forma geral do gerador polinomial é :

$$g(x) = (x - a^{i})(x - a^{i+1})...(x - a^{i+2t})$$

e a palavra código é construída usando;

$$c(x) = g(x) * i(x)$$

Onde *g(x)* é o gerador polinomial, *i(x)* é o bloco de informação, c(x) corresponde a uma palavra código válida e *a* corresponde ao elemento primitivo do campo. [COM04]

Exemplo: gerador para o RS (255,249)

$$g(x) = (x - a^{0})(x - a^{1})(x - a^{2})(x - a^{3})(x - a^{4})(x - a^{5})$$
$$g(x) = x^{6} + g_5 x^{4} + g_4 x^{4} + g_3 x^{3} + g_2 x^{2} + g_1 x^{1} + g_0$$

Os símbolos de paridade 2*t* numa palavra código sistemática de Reed-Solomon são dados por:

$$p(x) = i(x).x^{n-k} \bmod g(x)$$

A figura 4.5 mostra uma arquitetura de um codificador sistemático RS (255,249):

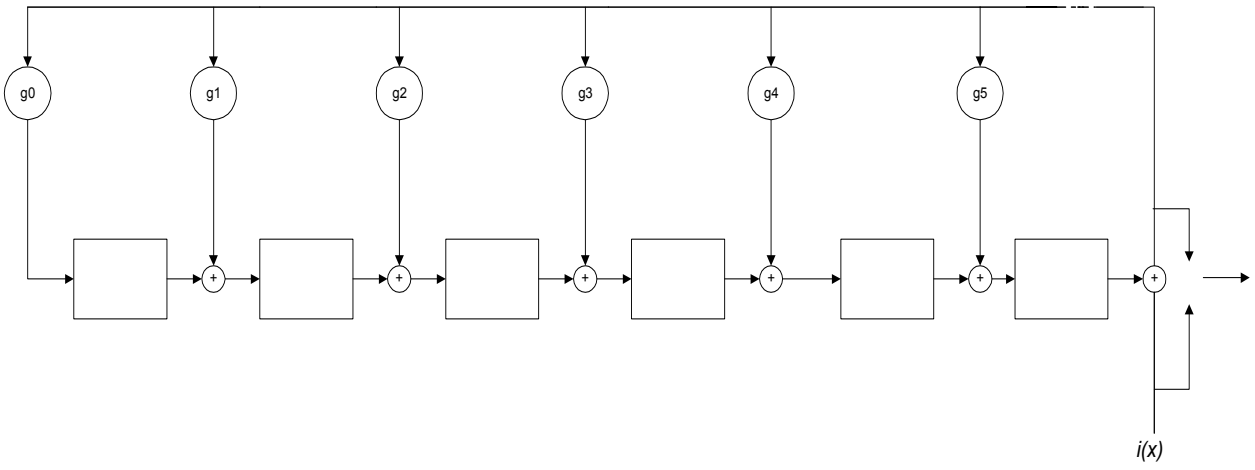

Fig 4.5 Arquitetura do codificador RS [COM04]

Cada um dos 6 registradores contém um símbolo (8 bits). Os operadores aritméticos realizam a adição ou a multiplicação finita do campo em um símbolo completo.[COM04]

Uma arquitetura geral para decodificar códigos Reed-Solomon é apresentada no diagrama a seguir:

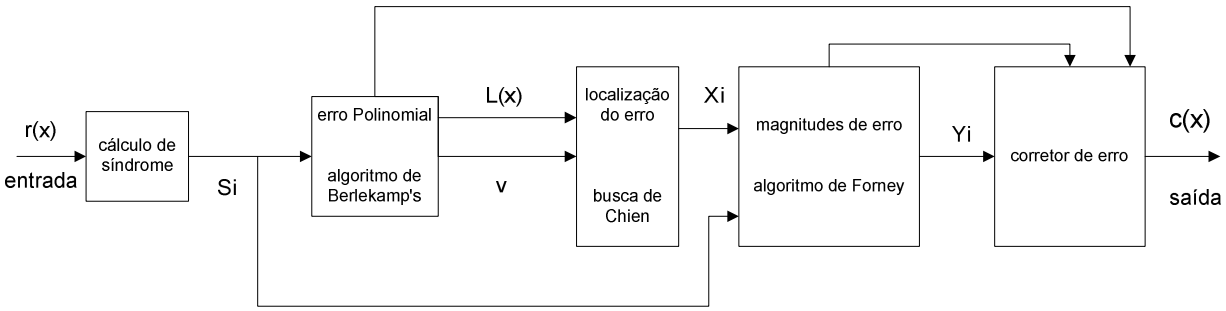

Fig 4.6 Arquitetura do decodificador RS [COM04]

Onde:

r(x) Palavra código recebida

Si Síndromes

L(x) Localizador de erro polinomial

Xi Localizações de erro

Yi Magnitudes de erro

c(x) Palavra código recuperada

v Número de erros

A palavra código recebida *r(x)* corresponde a palavra código original (transmitida) *c(x)* mais erros:

$$r(x) = c(x) + e(x)$$

Um decodificador Reed-Solomon procura identificar a posição e a magnitude de até t erros (ou $2t$ exclusões) e corrigir os erros ou exclusões. [COM04]

O cálculo de Síndrome é similar ao cálculo de paridade. Uma palavra Reed-Solomon possui $2t$ síndromes que depende unicamente de erros (não na palavra código transmitida). As síndromes podem ser calculadas pela substituição de $2t$ raízes do gerador polinomial *g(x)* dentro de *r(x)*.

Para se encontrar a localização dos símbolos de erro se faz necessário a resolução simultânea de equações com t incógnitas. Vários algoritmos são capazes de fazer isto. Estes algoritmos tiram vantagem da estrutura especial de matriz do código Reed-Solomon e reduzem

expressivamente o esforço computacional requerido. Geralmente, dois passos estão envolvidos:

Encontrar o localizador polinomial e a raízes deste polinômio. Para encontrar o localizador polinomial é possível usar o algoritmo de Euclides ou Berlekamp-Massey. O algoritmo de Euclides tende a ser mais largamente utilizado pela sua facilidade de implementação. Contudo, o algoritmo de Berlekamp-Massey tende a dirigir-se para implementações de hardware e software mais eficientes. Para encontrar as raízes do polinômio, utiliza-se o algoritmo de busca de Chien. Novamente, para a resolução simultânea de equações com *t* incógnitas, um algoritmo rápido largamente utilizado é o algoritmo Forney.

Existem um número de implementações de codificadores e decodificadores Reed-Solomon. Muitos sistemas existentes usam circuitos integrados "off-the-shelf" que codificam e decodificam os códigos Reed-Solomon. Estes CIs tendem a suportar uma certa quantidade de programas (por exemplo, RS (255,k) onde t=1 até 16 símbolos). A proposta recente é avançar do VHDL ou Verilog (núcleo lógico ou núcleo intelectual proprietário). Estes possuem um número de vantagens sobre os CIs padrão. Um núcleo lógico pode ser integrado com outro VHDL ou componentes Verilog e sintetizados para um FPGA (Field Programmable Gate Array) ou ASIC (Applicatios Specific Integrated Circuit) – isto habilita o chamado "System on Chip", onde múltiplos módulos podem ser combinados em um único CI. Dependendo do volume de produção, núcleos lógicos podem freqüentemente fornecer custos baixos frente aos CIs padrão.

Até recentemente, as implementações de software em tempo real requerem maior potencial computacional para todos os códigos, com exceção dos códigos Reed-Solomon. A maior dificuldade em implementar códigos Reed-Solomon em software é que os processadores de propósito geral não suportam as operações aritméticas do campo de Galois. Por exemplo, para implementar uma operação de multiplicação usando campo de Galois em software, requer um teste para 0, duas tabelas de busca de registro de log. [COM04]

Contudo, um projeto detalhado em conjunto com melhorias no desempenho do processador, significa que as implementações podem ser operadas em taxas relativamente altas. A seguinte tabela fornece alguns exemplos de desempenho em um PC Pentium 166 MHz:

Tabela 4.3: Tabela de desempenho [COM04]

Código	Taxa de dados
RS (255,251)	12 Mbps
RS (255,239)	2.7 Mbps
RS (255,223)	1.1 Mbps

Estas taxas correspondem ao pior caso de decodificação (corrigindo o número máximo de erros em cada palavra código): a codificação é consideravelmente rápida desde que requeira menor processamento.

4.9 Conclusão

Neste capítulo discutimos os princípios de proteção desigual de erros, apresentamos as estratégias já propostas na literatura para os códigos de compressão LZ77 e LZW.

Iniciamos exemplificando os modelos de canal com e sem memória, e os tipos de erros que estes estão sujeitos ao serem transmitidos por fio, cabo ou via satélite. Em seguida, apresentamos estratégias de controle de erro tais como o FEC e ARQ. Nas subseções 4.4.1 e 4.4.2 descrevemos os algoritmos propostos para a recuperação de erros nos códigos LZW e LZ77. Na Seção 4.5,descrevemos a proteção desigual de erros onde nem todos os dados possuem peso igual para o usuário. Para facilitar a compreensão das propriedades dos códigos lineares, na Seção 4.7 inserimos os conceitos de grupos e campos.

Neste capítulo também apresentamos uma descrição do código Reed-Solomon, que é apropriado para a correção de surtos de erros. O código Reed-Solomon é um código de bloco linear com uma vasta gama de aplicações em comunicações digitais.

Observou-se que através da utilização de esquemas de proteção desigual de erros para os códigos LZW e LZ77, pode-se obter maior proteção nas partes mais importantes dos dados compactados [HEL96]. Esta mesma estratégia foi adotada em nosso modelo proposto para o algoritmo LZSS, cuja implementação e resultados estão descritos no Capítulo 5.

Capítulo 5

Análise de Resultados

A crescente demanda por tecnologias de transmissão de dados sem fio (wireless) com alta capacidade de transmissão e confiabilidade na entrega de informação, tem exigido o emprego de estratégias cada vez mais elaboradas para compressão e codificação de canal, de maneira a se fazer um uso eficiente dos recursos de banda disponíveis para estas tecnologias.

Os efeitos do multipercurso (desvanecimento) em canais de rádio implicam na ocorrência de erros em surtos, caracterizando um canal com memória. Conforme discutimos na Seção 4.4 do Capítulo 4, a ocorrência de surtos de erros em arquivos compactados tem efeitos significativos no processo de descompressão. O outro contexto importante de aplicação de estratégias de proteção desigual de erros em arquivos compactados é em processos de armazenagem ou gravação digital de dados. Esta estratégia possibilita uma maior recuperação parcial de dados corrompidos em dispositivos de gravação ótica ou magnética.

A utilização de métodos conjuntos de codificação fonte (compressão) e codificação de canal (detecção/correção de erros) [HEL96] possibilita o desenvolvimento de estratégias para contenção da propagação de erros em arquivos ou pacotes de dados compactados corrompidos pelo canal de comunicação ou meio de armazenagem. Portanto a aplicação de estratégias de proteção desigual de erro, descritas na Seção 4.3 do Cap. 4 tem fundamental importância.

Neste capítulo avaliamos o desempenho da estratégia de proteção desigual proposta no Capítulo 4 para diferentes níveis de proteção gerados pelos códigos corretores de erro empregados.

5.1. Metodologia Empregada

Da revisão de literatura sobre proteção desigual de erro, realizada no capítulo 4, selecionou-se o artigo de Fujiwara, que analisou o mesmo arquivo de dados, denominado "*paper1.txt*", empregando dois métodos de compressão e detecção de erros diferentes, LZW e LZ77 [FUJ00].

Para este estudo, foi escolhido o arquivo de dados "*paper4.txt*" [BEL90], para estabelecer uma relação entre o resultado obtido em [FUJ00] empregando os dois métodos de compressão, e o resultado adotando-se a metodologia deste estudo.

A metodologia adotada consistiu em utilizar o aplicativo MATLAB® para a análise da estrutura do arquivo compactado e do efeito dos surtos de erro sob cada um dos seguintes componentes do código de compressão: bits de flag, bits de offset, bits de match e bits de caractere, identificando onde o efeito dos surtos de erros provoca maior propagação.

Inicialmente foi utilizado o arquivo texto "*paper1.txt"* [FUJ00] com 53 Kbytes. Foram realizados testes preliminares de inserção de erros de 8, 16, 64 e 96 bits, utilizando o programa desenvolvido no MATLAB®, apresentado no Apêndice I. O programa gera um vetor, que registra para cada posição de inserção de erro, a quantidade de erros que ocorrem no arquivo descompactado em relação ao original. Foram gerados 10 vetores com diferentes resultados pois a inserção dos erros se fazia de forma randômica. Uma vez gerados os vetores, estabelecia-se uma média dos valores de cada posição, gerando-se uma matriz final de erro. Constatou-se que o tempo para a extração de um único resultado atingiu uma média de 48 horas.

O modelo de erro aplicado teve como parâmetros surtos de 48 bits e surtos de 96 bits no arquivo texto *paper4.txt* [BEL90], para minimizar o tempo de execução do programa desenvolvido no aplicativo MATLAB®.

A adoção do código de compressão LZSS deve-se ao fato deste pertencer a um grupo de códigos em que a estatística da fonte, ou o conhecimento prévio da fonte a ser compactada, não se faz necessário. Por este motivo, Lempel-Ziv usou o termo "algoritmo universal" [ZIV77] em seu artigo que definiu o algoritmo LZ77 [HEL96]. A descrição detalhada do código LZSS, ou código de dicionário adaptativo, foi apresentada no Capítulo 3.

5.2. Análise da Estrutura dos Arquivos Compactados

Conforme mostrado na Figura 5.1, o programa no aplicativo MATLAB® possibilita a análise da estrutura do arquivo compactado. O arquivo de entrada "*paper4.txt*" é compactado pelo programa de compressão LZSS. Ao se efetuar a leitura deste arquivo, temos um vetor em decimal em códigos ASCII. Este vetor é convertido para binário e as informações contidas neste arquivo compactado são segmentadas em quatro componentes: bits de flag, bits de caractere, bits de match, bits de offset. Para o código LZSS, existe um flag indicador da necessidade de compressão ou não dos dados. Durante a segmentação, a separação dos diferentes componentes é feita em função da leitura do flag. Se for 0, a informação subseqüente é não codificada, se for 1, a informação subseqüente é uma referência para o dicionário. Esta referência é composta da correspondente posição do símbolo no dicionário (offset) e do comprimento da string (matchbits). Enquanto a diferença entre o tamanho do vetor e o contador for maior ou igual a 8 bits, o processo de segmentação terá continuidade. Quando a diferença for menor do que 8 bits, haverá uma interrupção no processo de segmentação nos quatro componentes, e uma nova fase de programação será iniciada. Para cada análise do efeito de propagação de surtos de erros nos diferentes segmentos, a etapa descrita anteriormente e ilustrada na Figura 5.1 se repetirá.

Foi elaborado um programa específico no MATLAB®, que realiza a inserção do surto de erros para cada componente (flag, caractere, matchbits, dicbits). Um programa adicional foi elaborado no MATLAB® para uma análise geral da propagação de erros no arquivo, sem a segmentação. Neste último caso, o surto de erros poderá afetar qualquer parte da estrutura do arquivo compactado e isto implicará em uma propagação de erro completamente diversa da inserção de erros em um dos componentes, conforme exposto na subseção 5.2.1, onde não houve a segmentação dos dados.

As subseções 5.2.1 à 5.2.4 ilustram os resultados obtidos mediante os programas de inserção de surtos de erros, sobre cada componente (flag, caractere, matchbits, dicbits).

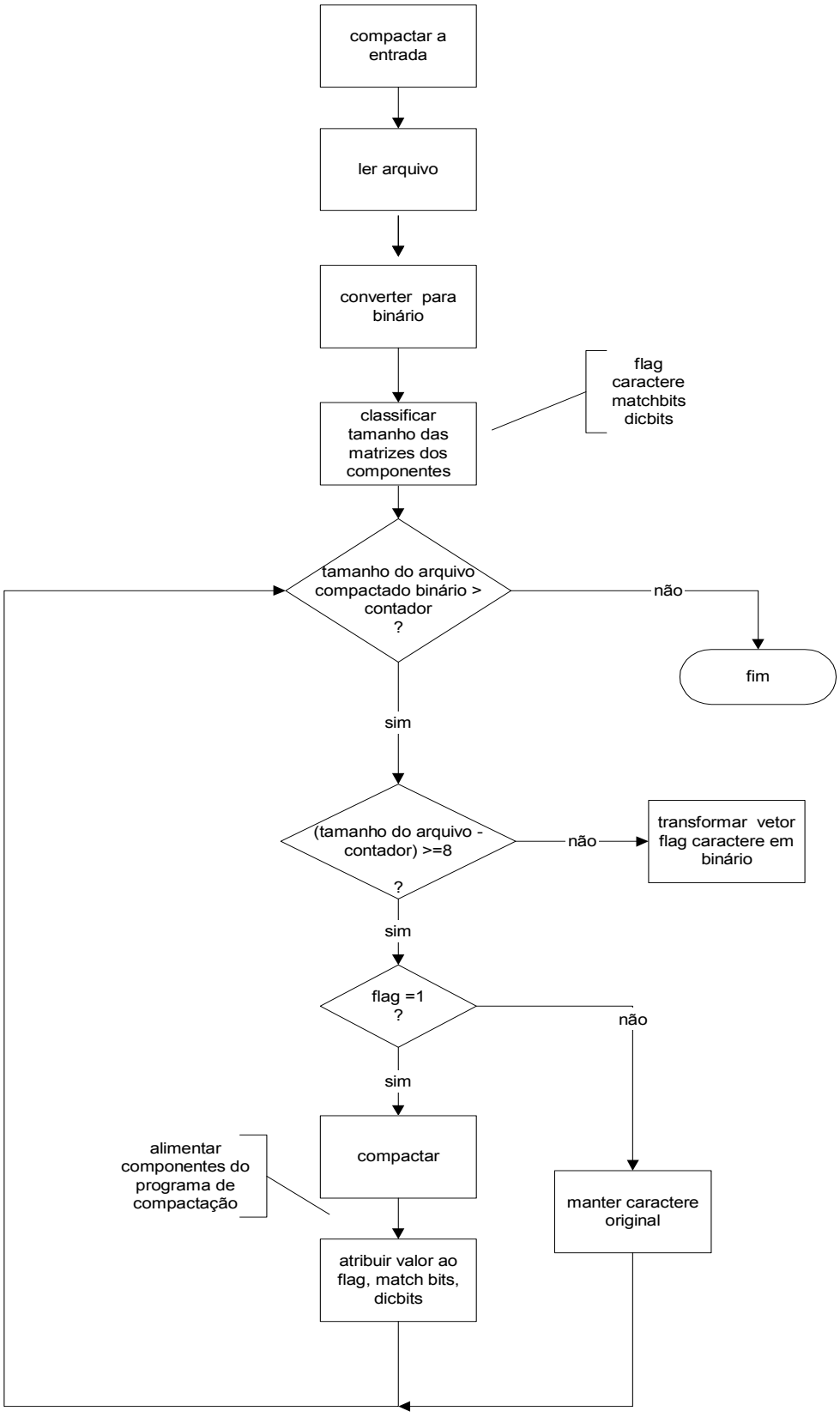

Fig 5.1 – Fluxograma para o programa da decomposição do arquivo compactado

Os testes se basearam em analisar através da segmentação dos componentes utilizados no código de compressão LZSS do arquivo ainda compactado, o efeito da propagação do erro no processo de descompactação. Após a descompactação do arquivo, realiza-se a contagem do número de erros ocorridos no arquivo descompactado comparando-o ao arquivo original.

Para o tamanho da janela de compressão que compreende o "search buffer - dicionário" e o look-ahead buffer foram adotados valores padrões utilizados pelo algoritmo , 16 e 12 respectivamente [HEL96]. À medida que se aumenta o tamanho do dicionário, a probabilidade de se aumentar o casamento de uma string entre o dicionário e o look-ahead buffer é maior. No entanto, qualquer aumento no comprimento do dicionário, resultará no aumento do número de comparações de strings que deverão ser realizadas no look-ahead buffer [HEL96]. Portanto, para este código em particular, será exigido um período maior para a compactação de um determinado arquivo se comparado ao mesmo código de tamanho de dicionário inferior. O tamanho do lookahead buffer, comparado com o tamanho do dicionário é muito menor, contendo de 10 a 20 símbolos. Se o lookahead buffer for muito pequeno, strings mais longas não poderão ser codificadas eficientemente. Se for muito extenso, o tempo gasto para se determinar longos casamentos de string, quando estes podem até não existir, será muito maior. No LZSS, se for duplicado o tamanho do dicionário e utilizado uma árvore binária para armazenar os dados, o tempo requerido para localizar o casamento de string mais distante na árvore, resultará em um pequeno incremento no tempo de processamento, mesmo havendo esta duplicação no tamanho do dicionário [HEL96].

O arquivo compactado pode ser transmitido de acordo com a estrutura mostrada na Figura 5.2, onde *F* denota flag, *C* denota caractere, *M* corresponde ao matchbits, e *O* corresponde ao offset bits. Esta estrutura permite a descompactação direta do arquivo compactado como uma função dos dados recebidos. A partir da leitura seqüencial do arquivo compactado, conforme a estrutura da Figura 5.2, o algoritmo de descompactação vai reconstruindo o arquivo original. Quando é encontrado um bit de flag, *F=0*, os 8 bits subseqüentes são mapeados no caractere estendido ASCII [HEL96] correspondente. Quando for encontrado um bit de flag, *F=1*, os 16 bits de offset subseqüentes indicam o ponto no arquivo já descompactado até o momento, a partir do qual deve ser copiada uma seqüência de caracteres, cujo comprimento é indicado pelos 12 bits de match que seguem os 16 bits de offset. A estrutura do arquivo para esquemas de codificação de canal com proteção desigual de erro é mostrada na Figura 5.3, onde agrupamentos de bits com diferentes classes, podem

ser designados diferentes níveis de proteção. Nesta estrutura, o número de bits em cada classe deve ser transmitido com o arquivo compactado. Esta informação (número de bits em cada classe) constitui o cabeçalho mostrado na Figura 5.3. Neste estudo, consideramos a transmissão do cabeçalho livre de erros.

Fig. 5.2. Estrutura do arquivo compactado.

Fig. 5.3. Estrutura do arquivo compactado para proteção desigual de erro.

Na Tabela 5.1 são apresentados os parâmetros adotados no programa de compactação. O programa LZSS adotado para a compactação dos dados para este estudo foi desenvolvido por Rich Geldreich, Jr., disponibilizado no site www.programmersheaven.com [GEL93]. Outro programa em linguagem C para compactar o arquivo antes de submetê-lo à segmentação realizada pelo MATLAB®, está sugerido por Held, em seu livro Data and Image Compression [HEL96]. Especificamente, este estudo apresenta resultados para o arquivo de texto "*paper4.txt*" de tamanho 13 Kbytes. A Tabela 5.2 resume as características do arquivo compactado.

Tabela 5.1. Parâmetros utilizados no algoritmo de compressão LZSS

Parâmetro	**Tamanho**
Bits de *Flag*	1 bit
Bits de *Caractere*	8 bits
Bits de *Match*	12 bits
Bits de *Offset*	16 bits

Tabela 5.2 Dados do Arquivo Compactado

ESPECIFICAÇÃO	TAMANHO
Tamanho do arquivo não compactado	13286 bytes
Tamanho do arquivo compactado	9014 bytes
Número total de bits no arquivo compactado	72108 bits
Número total de símbolos de flag no arquivo compactado	3772 símbolos
Número de flag bits no arquivo compactado	3772 bits
Número de símbolos de caracteres no arquivo compactado	1864 símbolos
Número de bits de caracteres no arquivo compactado	14912 bits
Número de símbolos de dicionário no arquivo compactado	1908 símbolos
Número de bits de dicionário no arquivo compactado	30528 bits
Número de símbolos de match no arquivo compactado	1908 símbolos
Número de match bits no arquivo compactado	22896 bits

Os histogramas nas Figuras 5.4 e 5.5 mostram a distribuição dos bits de match e dos bits de offset, respectivamente, no arquivo compactado levando-se em consideração os parâmetros inicialmente especificados na Tabela 5.1. Na Figura 5.4 tem-se o componente de match bit que constitui parte da referência de dicionário. Como já especificado neste mesmo capítulo, os bits de match correspondem ao comprimento da seqüência de caracteres que deverá ser copiada a partir de uma dada posição (offset) do arquivo descompactado. Ainda na Figura 5.4, observa-se para um dado comprimento o número de caracteres correspondentes dentro do arquivo compactado. Conforme podemos observar da Figura 5.5, a referência do dicionário tende a aumentar o número de caracteres compactados no decorrer do processo de compressão. Isto se dá em função da maior ocorrência de "casamentos" das seqüências de caracteres.

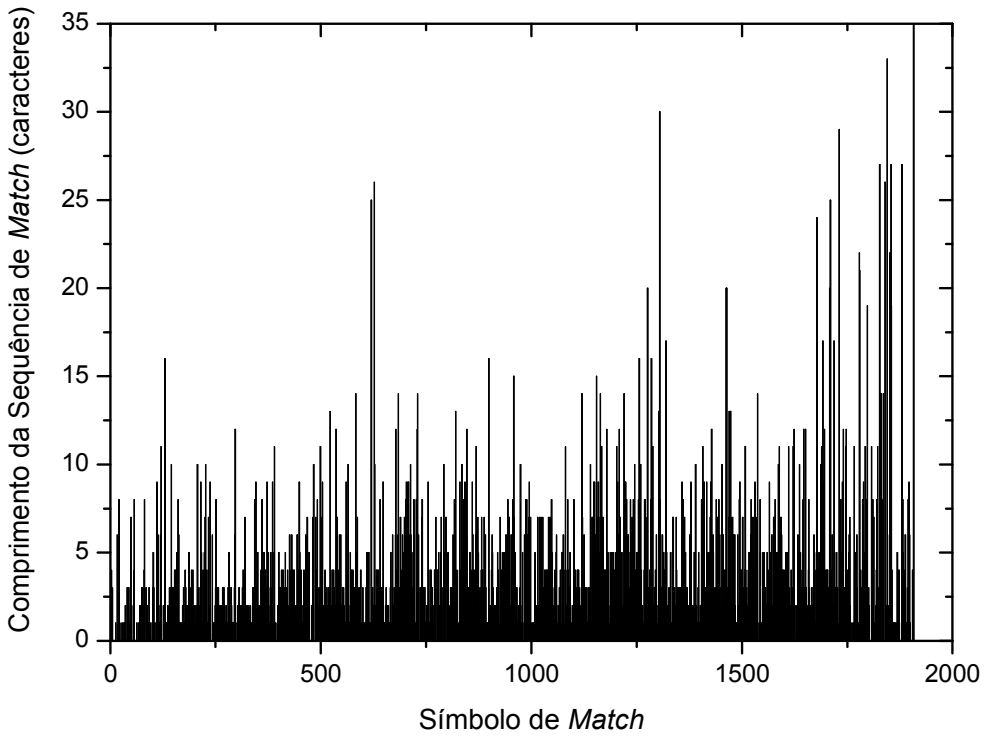

Fig. 5.4. Estrutura do arquivo compactado para proteção desigual de erro.

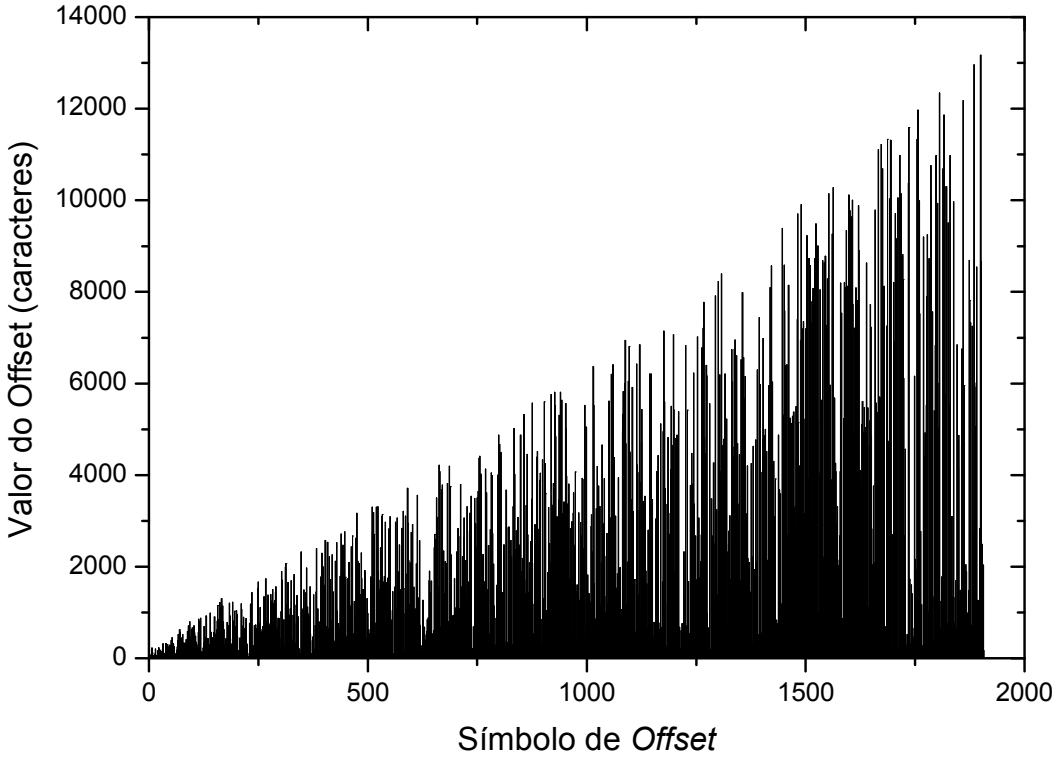

Fig.5.5. Histograma dos símbolos de dicionário no arquivo compactado.

Houve uma segmentação do arquivo dos componentes utilizados no código de compressão LZSS. Os componentes eram *bits de flag*, *bits de caractere*, *bits de match* e *bits de offset* (dicionário), conforme descritos no Capítulo 4. Na compressão do arquivo *paper4.txt*, o programa de compressão LZSS utiliza os componentes acima indicados para realizar este processo. Para se conseguir uma melhor visualização do efeito da propagação do erro sobre o arquivo original após a sua descompactação, surtos de erro em cada um dos segmentos foram inseridos. Os seguintes testes foram realizados:

- Inserção de surtos de erro sobre o arquivo compactado não segmentado;
- Inserção de surtos de erro no arquivo compactado com entrelaçamento;
- Inserção de surtos de erro sobre os *bits de flag* do código LZSS no arquivo compactado;
- Inserção de surtos de erro sobre os *bits de match* do código LZSS no arquivo compactado;
- Inserção de surtos de erro sobre os *bits de offset* do código LZSS no arquivo compactado;

- Inserção de surtos de erro sobre os *bits de caractere* do código LZSS no arquivo compactado;
- Teste do efeito da inserção de surtos de erro sobre o arquivo compactado não segmentado utilizando códigos corretores de detecção e correção de erro do tipo Reed-Solomon.
- Comparação entre surtos de erro de 48 bits e 96 bits sobre os bits de flag

Neste procedimento de teste, primeiro foi compactado o arquivo "*paper4.txt*" [BEL90] usando o algoritmo LZSS, conforme já citado na Seção 5.1. O arquivo compactado é processado pelo programa, que realiza a decomposição em seus componentes: flag, caracter, offset bits e match bits. Na seqüência, o efeito da propagação do erro é analisado por cada parâmetro independentemente. Segundo o procedimento adotado em [BEL90] e [FUJ00], aplicou-se um surto de erro de 48 bits para o arquivo compactado. Mediante este procedimento, é possível identificar a influência dos erros sobre cada parâmetro do arquivo compactado e seu impacto na reconstrução do arquivo original durante o processo de descompactação.

5.2.1 Impacto dos Surtos de Erro no Arquivo Compactado Não Segmentado

Na primeira análise mostrada na Figura 5.6, avaliamos o impacto de um surto de erro de 48 bits introduzido em diferentes posições do arquivo compactado. A curva representa o percentual de erros no arquivo descompactado em função da localização do surto de erro no arquivo compactado. Para esta análise não houve a segmentação dos dados. O arquivo "*paper4.txt*" foi compactado e foram inseridos surtos de 48 bits ao longo de todo o arquivo. O eixo *y* representa o percentual de erros no arquivo descompactado como uma função da localização do erro no arquivo compactado. A unidade da localização do erro (eixo *x*) é em caractere (8 bits símbolos). Isto é importante para apontar que a propagação do erro é mais significativa se os surtos de erros ocorrerem no início do arquivo compactado, pois é nesta posição que se encontra o dicionário. Portanto, a informação do buffer de busca torna-se inconsistente, provocando a propagação do erro durante a descompactação.

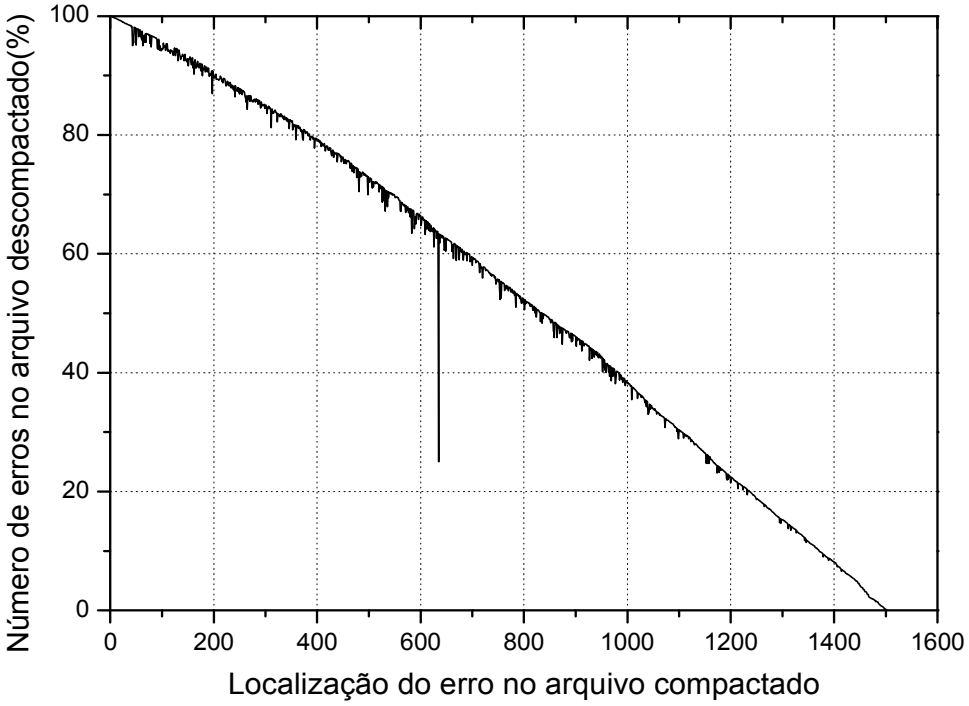

Fig 5.6 Efeito dos surtos de erro no arquivo compactado não segmentado

5.2.2 Impacto do Surto de Erros no Arquivo Compactado com Entrelaçamento

A Figura 5.7 apresenta o efeito do surto de erros quando o arquivo compactado sofre o um entrelaçamento de bits antes de ser transmitido ou armazenado. A técnica de entrelaçamento é comumente utilizada em sistemas de transmissão ou armazenagem digital para minimizar o efeito de memória do canal, ou seja, espalhar eventuais surtos de erros ao longo do arquivo compactado. Com o objetivo de avaliar esta técnica em arquivos compactados, avaliamos o impacto dos surtos de erros num arquivo compactado entrelaçado. Foi utilizada uma estrutura de entrelaçamento uniforme usando matriz, com escrita em linhas e leitura em colunas. A profundidade do entrelaçamento utilizado foi de 36056 bits, o que significa que dois bits subseqüentes do arquivo original irão estão distantes entre si de 36056 bits após o entrelaçamento. Podemos observar que a função de entrelaçamento dissemina os surtos de erro ao longo do arquivo compactado e amplifica o efeito de propagação do erro no processo de descompactação.

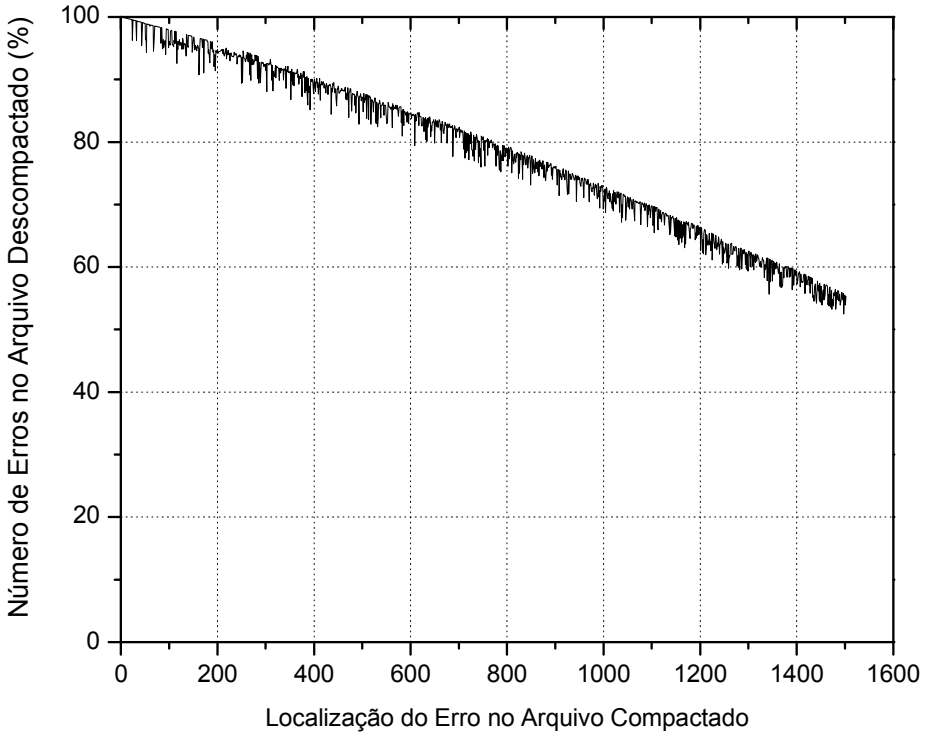

Fig.5.7 Efeito dos surtos de erro no arquivo compactado entrelaçado.

5.2.3 Impacto do Surto de Erros nos Bits de Flag do Arquivo Compactado

A inserção de surtos de erro de 48 bits no arquivo compactado não segmentado, conforme estrutura mostrada na Figura 5.2, produz um efeito similar a corrupção apenas dos flags do mesmo arquivo quando segmentado. A Figura 5.8 apresenta o efeito da propagação do erro no arquivo descompactado considerando a ocorrência de surtos de erro apenas nos bits de flag. Para esta análise, considerou-se 12 bits de match bits e 16 bits de dicionário. Como descrito na seção anterior, o flag indica se o símbolo subseqüente corresponde a um texto não codificado ou referência de dicionário (palavra código). Se o bit de flag é corrompido temos como conseqüência a substituição do texto por palavras código e vice-versa. Isto resultará em bits de offset indicando posições de inicialização incorretas ou incompatibilidade de tamanhos no dicionário. A propagação do erro gerada pelos flags é similar aos resultados apresentados na Figura 5.7, onde os erros são introduzidos seqüencialmente no arquivo compactado afetando os bits de flag, os bits de offset, os match bits e os bits de caractere. Os erros nos bits de flag corrompem o tamanho do arquivo no processo de descompactação, resultando em um arquivo descompactado com tamanho diferente do original.

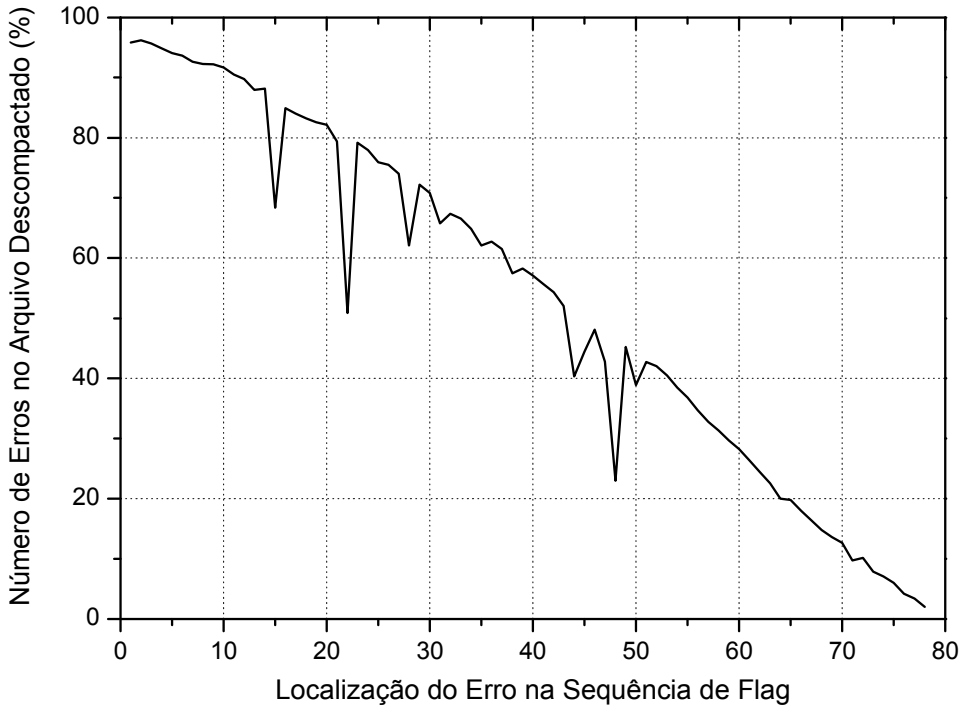

Fig 5.8 Efeito dos surtos de erro na seqüência de bits de flag do arquivo compactado com 12 bits de match bits e 16 bits de dicionário

5.2.4 Impacto do Surto de Erros nos Bits de Match do Arquivo Compactado

Os bits de match também são importantes no procedimento de descompactação conforme se observa na Figura 5.9. O parâmetro de bit de match corresponde a um dos componentes de referência do dicionário. Os bits de match indicam o número de caracteres que deverão ser copiados no arquivo descompactado a partir da posição informada pelo offset, outro componente da referência do dicionário. Os surtos de erro nos bits de match provocam a reprodução incorreta do número de caracteres durante o processo de descompactação. Para a análise da Figura 5.9, estão sendo considerados surtos de erro de 48 bits com 12 bits de offset de 14 bits de match.

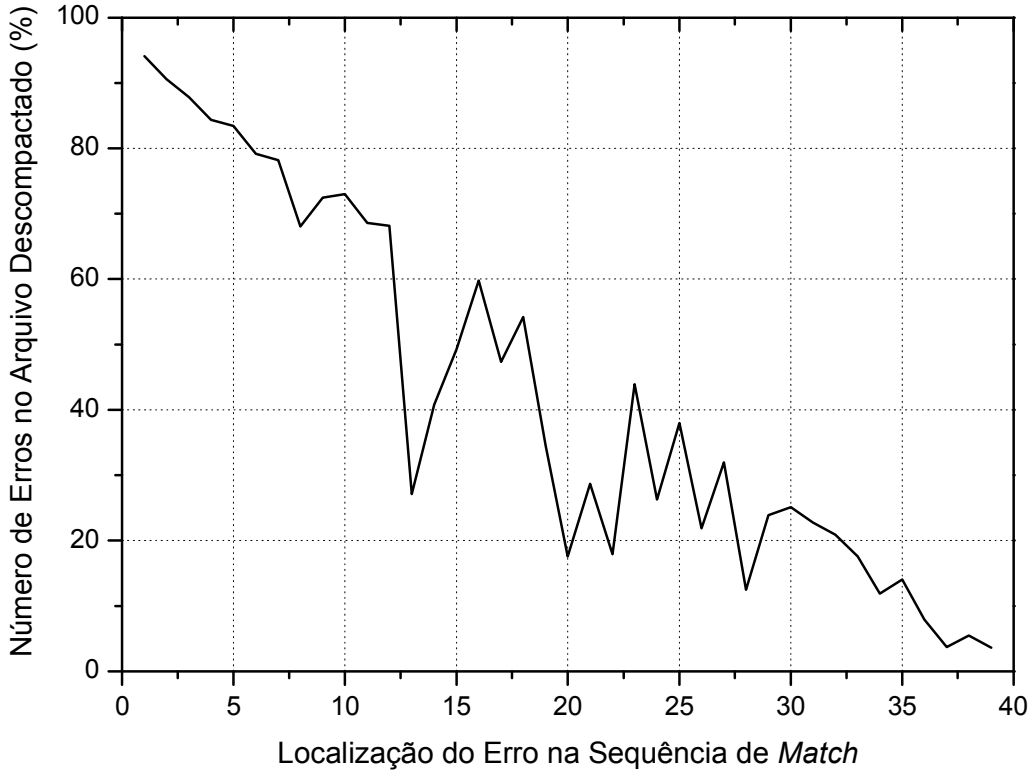

Fig. 5.9 Efeito do surto de erro sobre a seqüência de match bits do arquivo compactado com 12 bits de match bits e 16 bits de dicionário

5.2.5 Impacto do Surto de Erros nos Bits de Offset do Arquivo Compactado

Nas Figuras 5.10 e 5.11 observam-se como resultado dos surtos de erros de 48 bits com um dicionário (offset) de tamanho de 12 bits e uma janela móvel (matchbits) de 14 bits e 4 bits, respectivamente. A propagação de erro no arquivo descompactado é inferior a 25% e 5%, respectivamente. Este parâmetro é responsável pela indicação da posição no arquivo descompactado em que se iniciará a cópia dos caracteres durante o processo de descompactação. Com surtos de erro neste parâmetro, a referência do dicionário estará mais uma vez comprometida. O erro se propagará, pois outras seqüências de caracteres no processo de reconstrução do arquivo descompactado substituirão a original.

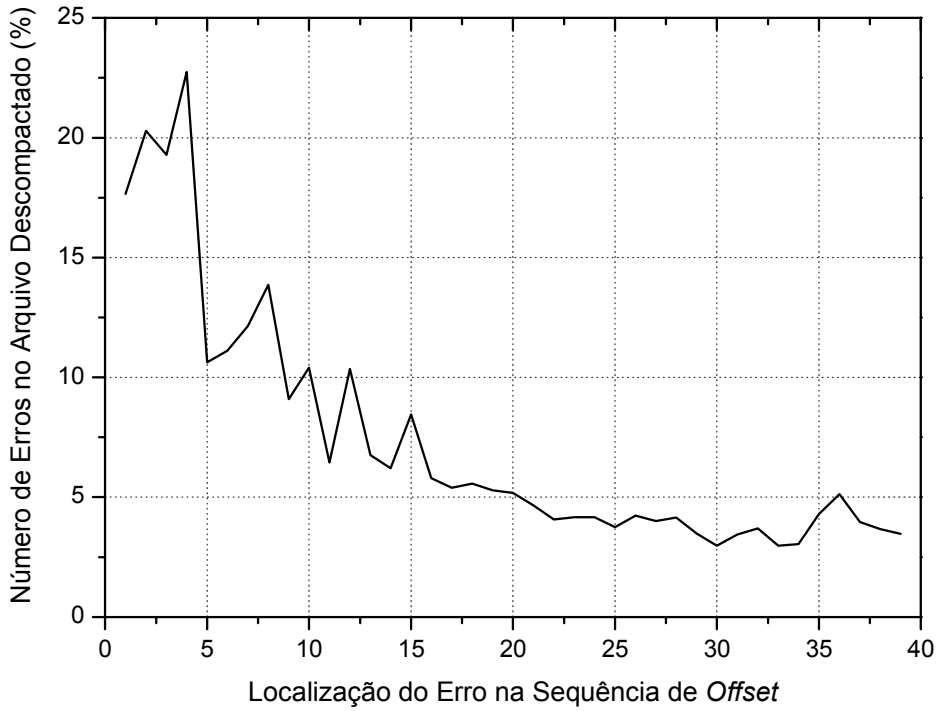

Fig 5.10 Efeito dos surtos de erro sobre os bits de offset no arquivo compactado com 12 bits de match bits e 16 bits de dicionário

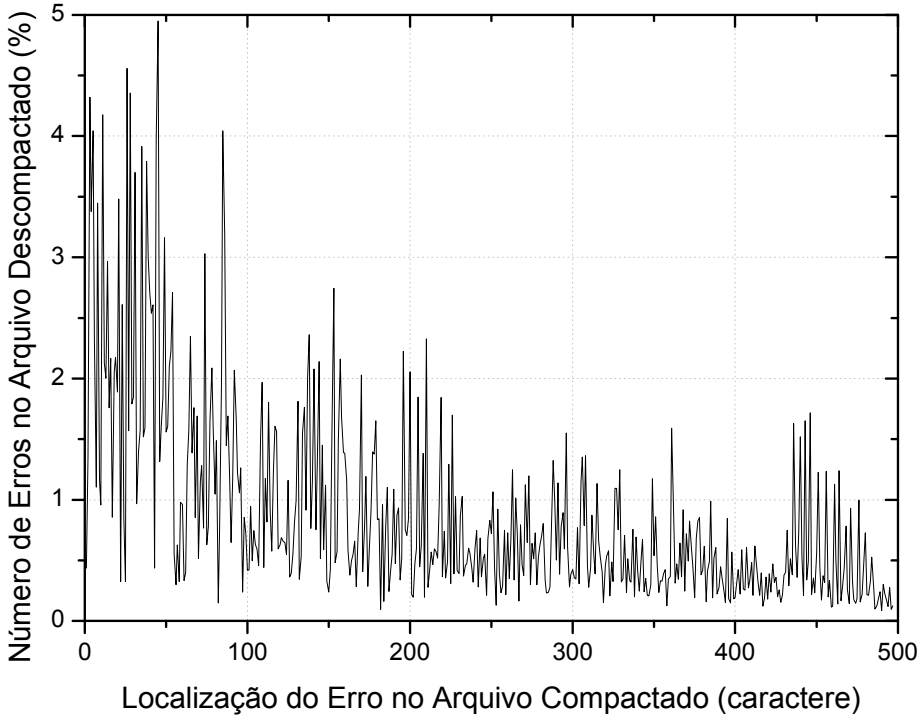

Fig 5.11 Efeito dos surtos de erro sobre os bits de offset no arquivo compactado com 4 bits de match bits e 12 bits de dicionário

5.2.4 Impacto dos Surtos de Erros nos Bits de Caractere do Arquivo Compactado

As Figuras 5.12 e 5.13 representam o impacto do deslocamento do surto de erros de 48 bits com bits de match de 12 e 4 bits respectivamente, ao longo da seqüência de caracteres do arquivo compactado. A propagação do erro é inferior a 5% em ambos os casos porque ela é confinada a cada caractere, não sendo distribuída ao longo do arquivo. O impacto gerado na descompactação é mínimo, podendo o arquivo ser recuperado com uma pequena fração de erros em relação ao arquivo original descompactado.

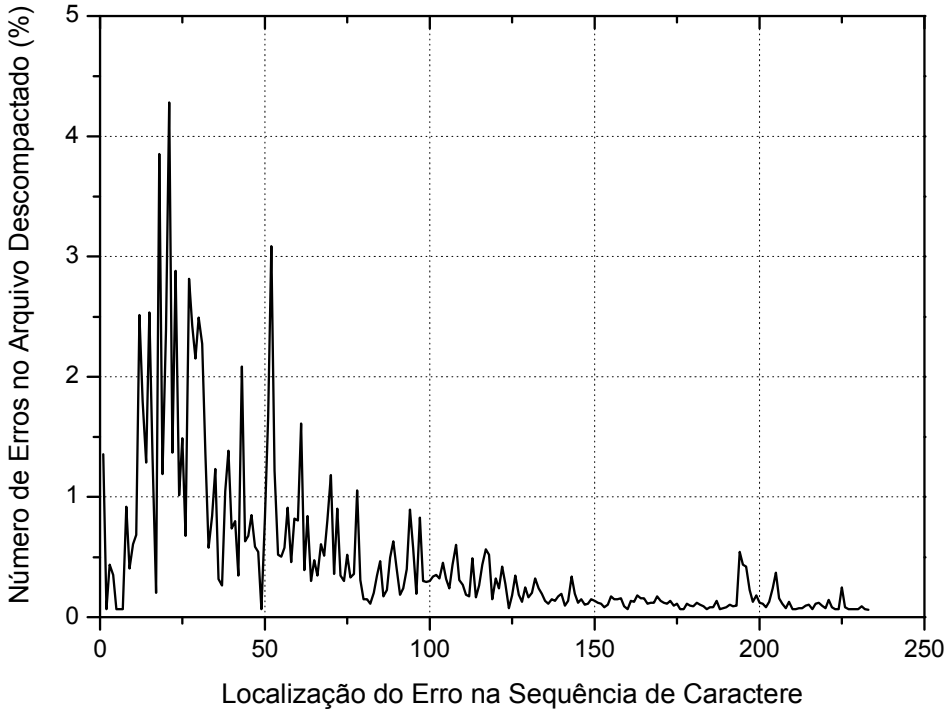

Fig 5.12 Efeito dos surtos de erro na seqüência de bits de caractere do arquivo compactado com 12 bits de match bits e 16 bits de dicionário.

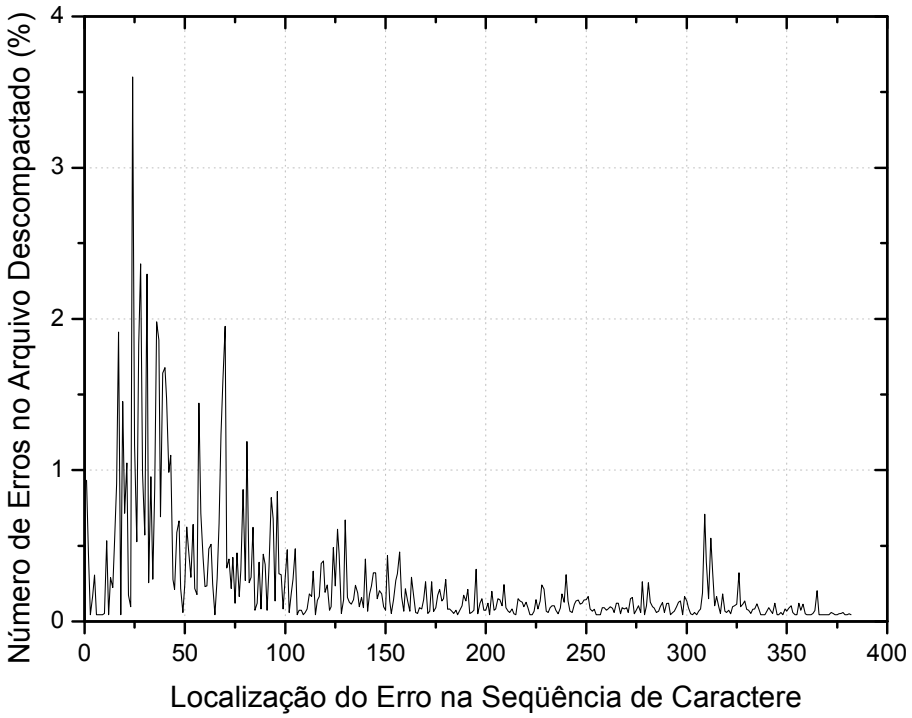

Fig 5.13 Efeito dos surtos de erro na seqüência de bits de caractere do arquivo compactado com 4 bits de match bits e 12 bits de dicionário.

Baseando-se nesta análise, a estratégia de proteção de erro que será aplicada aos dados compactados pode ser otimizada. Erros nos bits de dicionário e nos bits de caractere causam menos impacto no processo de descompactação. No entanto, erros nos bits de flag e nos bits de match causam um efeito drástico na descompactação, e ainda mais severo se ocorrerem no início do arquivo compactado.

Outros resultados obtidos comparando-se surtos de erro de 48 bits e de 96 bits são apresentados nas Figuras 5.14 à 5.17. Observa-se que a medida que os surtos de erro aumentam, a propagação dos erros ao longo do arquivo compactado também aumenta, gerando arquivos de tamanhos maiores que os originais.

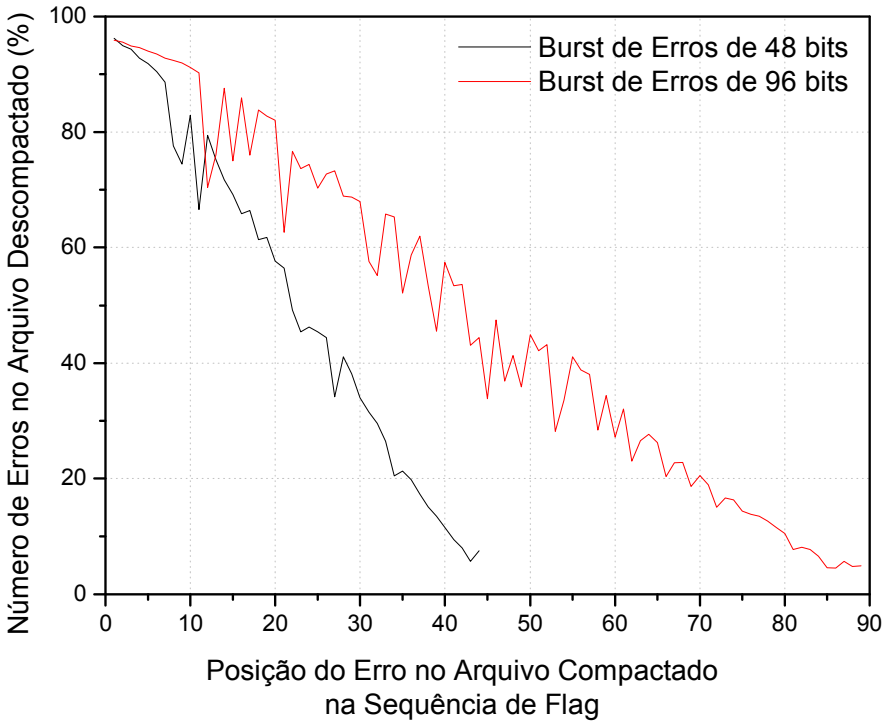

Fig. 5.14 Comparativo entre surtos de erro de 48 bits e 96 bits entre as seqüências de flag com 4 bits de match bits e 12 bits de dicionário.

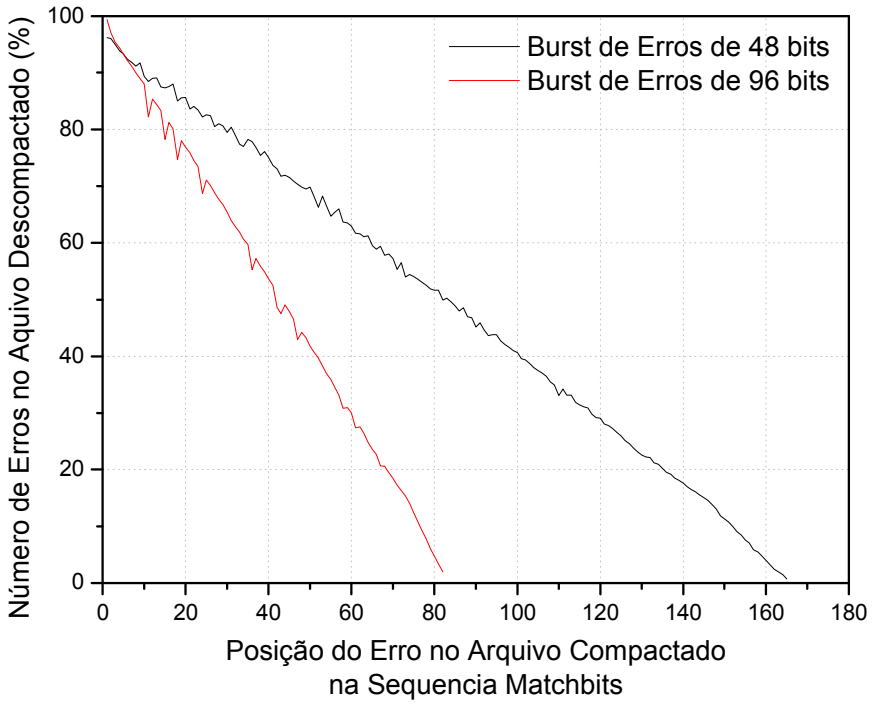

Fig. 5.15 Comparativo entre surtos de erro de 48 bits e 96 bits entre as seqüências de bits de match com 4 bits de match bits e 12 bits de dicionário.

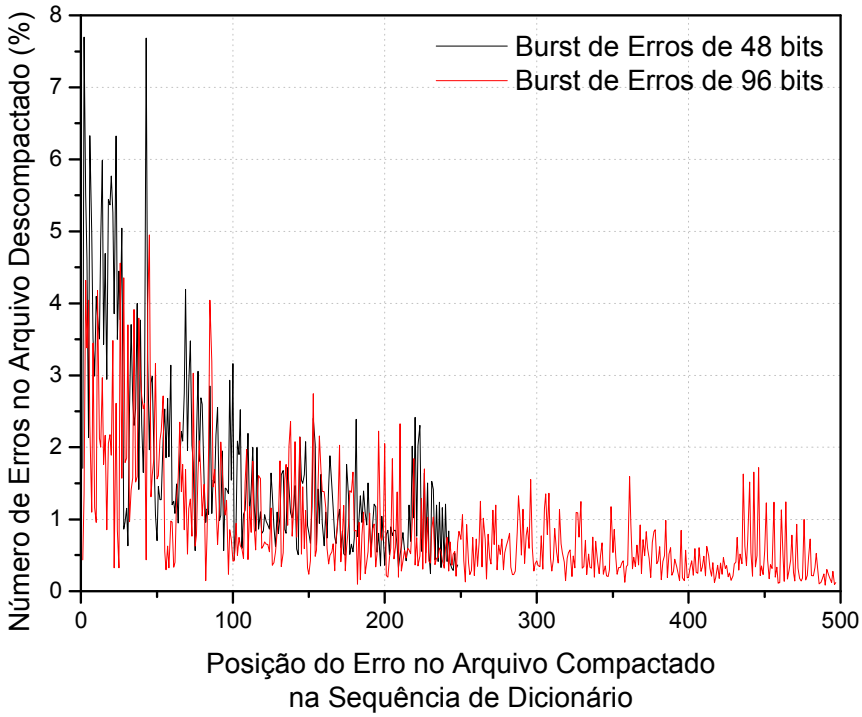

Fig. 5.16 Comparativo entre surtos de erro de 48 bits e 96 bits entre as seqüências de bits de dicionário (offset) com 4 bits de match bits e 12 bits de dicionário.

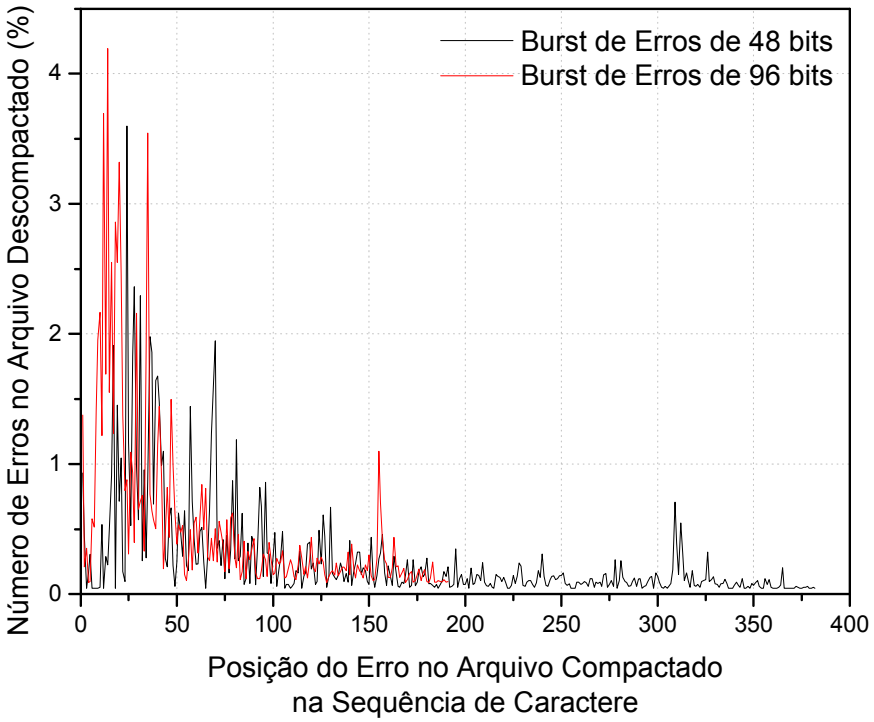

Fig. 5.17 Comparativo entre surtos de erro de 48 bits e 96 bits entre as seqüências de caractere com 4 bits de match bits e 12 bits de dicionário

5.3. Proteção Desigual de Erros

O esquema de proteção desigual de erro adotado tem por objetivo garantir maior confiabilidade durante a transmissão ou armazenamento dos dados. A proposta considera a adoção de um código de proteção e correção de erro para um dos parâmetros pertencentes ao código de compressão LZSS e compreendem: proteção geral, bits de flag, bits de match, bits de offset e bits de caractere. Cada um desses componentes quando tratados ou protegidos isoladamente, oferecem maior ou menor grau de proteção dos dados codificados quando utilizados na transmissão ou no armazenamento das informações.

A estratégia básica para fornecer proteção desigual contra erros é a utilização de esquemas de correção diferenciados para as classes de informação, em função do grau de importância estipulado para cada classe. No caso de dados compactados utilizando o algoritmo LZSS, avaliamos na Seção 3 que erros nos bits de flag tem maior impacto na descompressão, seguido dos bits de offset, bits de match e bits de caractere.

O codificador do Reed-Solomon se utiliza de um bloco de dados digital adicionando redundância extra de bits (redundância controlada). Com a ocorrência de erros no canal durante a transmissão o Reed-Solomon processa cada bloco no decodificador procurando corrigir os erros e recuperar os dados originais. O número e o tipo de erros que podem ser corrigidos dependem das características do código RS.

A Figura 5.18 ilustra as principais fases do código implementado no MATLAB® para proteção dos *bits de flag* no arquivo compactado. Apesar de representar a inserção de surtos de erro especificamente sobre os bits de flag, pertencente a um dos componentes do código de compressão adotado, este mesmo fluxograma pode também ser aplicado para outros componentes do código LZSS empregado: caractere, matchbits e dicbits. As diferentes fases do processo são definidas a seguir:

a) Primeira fase: ***Decomposição do arquivo compactado***

O código de compressão adotado para os testes foi decomposto em 4 componentes como já especificado anteriormente. Através da análise de cada um destes componentes, chegou-se a conclusão que os bits de flag são responsáveis pelo maior índice de propagação de erro. Por isso, com a proteção diferenciada dos bits de flag, mesmo com a incidência de erros, estará assegurada uma melhor recuperação dos dados após serem transmitidos ou armazenados.

b) Segunda fase***: Ajuste da dimensão da matriz para aplicar o código Reed-Sollomon***

Ajuste da matriz para múltiplos de 8 bits, por se considerar cada caractere de comprimento igual a 8 bits. Nem todos os arquivos terão a mesma quantidade de caracteres e por isso a matriz deverá ser redimensionada para poder se adaptar a diferentes tamanhos do arquivo. A codificação de Reed-Solomon necessita operar com matrizes múltiplas de 8 que trabalha com $GF(q)$, onde $q = 2 \times m$, sendo $m = 8$ o valor adotado para os testes.

c) Terceira fase: ***Passagem de um vetor decimal para o tamanho da mensagem***

Nesta fase, tem-se a conversão das palavras de 8 bits para decimal, estas palavras em decimal serão utilizadas no vetor de mensagem *Kc* (tamanho do bloco de mensagem) do algoritmo Reed-Solomon.

d) Quarta fase: ***Codificação Reed-Solomon***

Tem-se a codificação do componente de Flag do arquivo compactado utilizando-se do código Reed-Solomon, sendo alguns dos seus parâmetros especificados a seguir:

$t = 4$ Capacidade de correção do código

$nc = q - 1$ Tamanho do bloco codificado

$kc = nc - 2 \times t$ Tamanho do bloco de mensagem

e) Quinta fase: ***Inserção de erro e decomposição do arquivo com erros***

Ocorre a inserção de surtos de erro no arquivo compactado protegido com o código de detecção e correção de erros Reed-Solomon. Em seguida tem-se a decomposição deste mesmo arquivo sobre a influência do código Reed-Solomon especificamente no componente de bits de flag. Para a análise considerou-se surtos de erro de 48 bits.

f) Sexta fase***: Novo ajuste do dimensionamento da matriz***

Com a inserção de surtos de erro no arquivo compactado, o vetor FLAG2 foi modificado, havendo a necessidade de um redimensionamento da matriz para múltiplos de 8 bits.

g) Sétima fase: ***Passagem de um vetor decimal para o tamanho da mensagem***

Após o redimensionamento da matriz os dados são convertidos para decimal, sendo o vetor resultante FLAG_DECIMAL_2 equivalente ao vetor de entrada. Este mesmo vetor sofre uma nova alteração para a mensagem de tamanho Kc.

h) Oitava fase: ***Decodificação do arquivo, construção de um novo vetor de flag binário, decodificação do arquivo e construção do novo vetor de FLAG binário***

O vetor FLAG_Decimal_2 sofre a conversão para a mensagem de tamanho *Kc*. Esta mensagem é decodificada e o novo vetor de flag binário será construído.

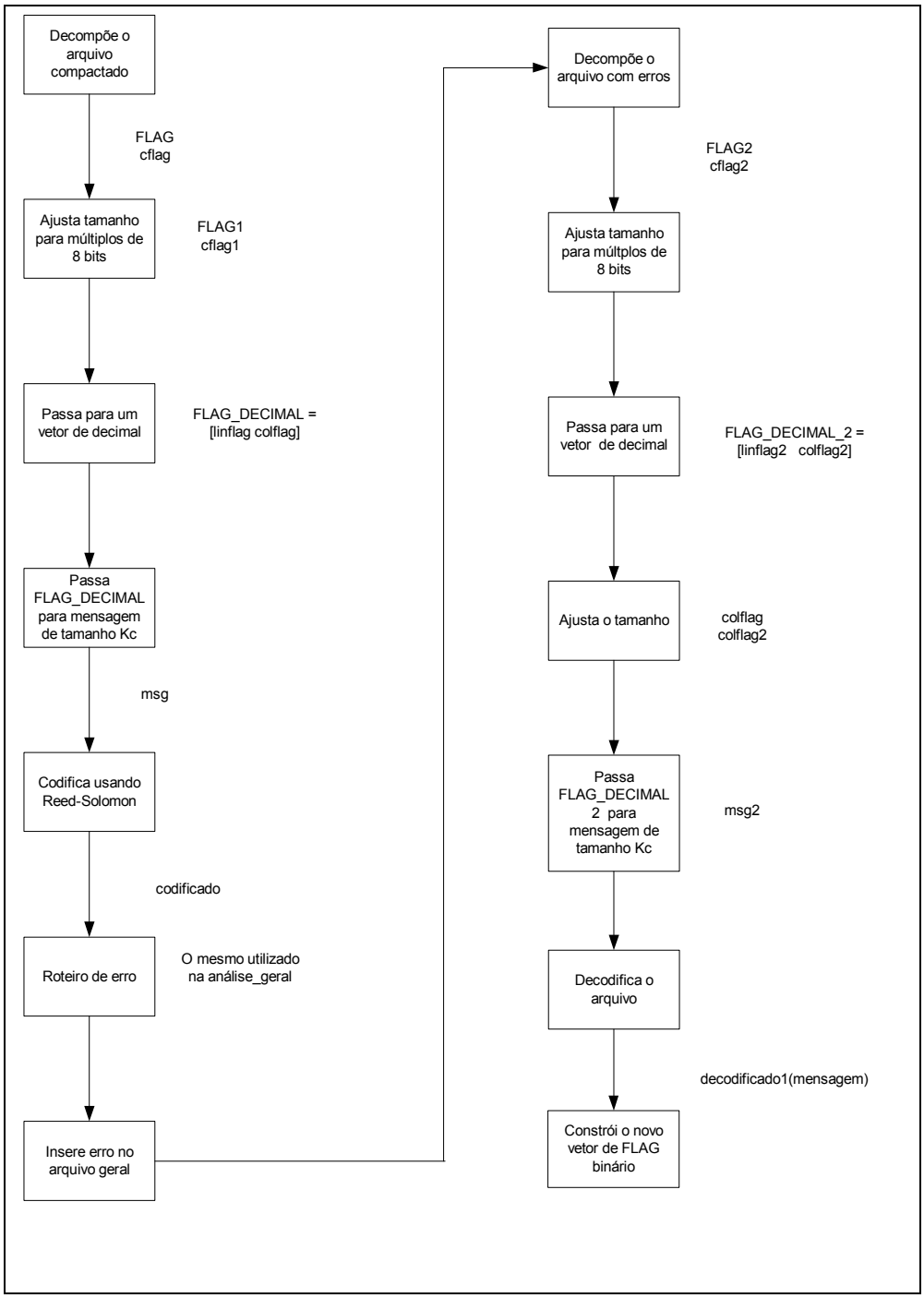

Fig.5.18 Algoritmo de codificação/ decodificação do arquivo utilizado para teste

5.3.1 Proteção Uniforme Contra Erros no Arquivo Compactado

Os códigos de bloco de Reed-Solomon são específicos para a correção de surtos em canais com memória, sendo portanto apropriados para correção de erros em algoritmos de dados compactados. A redundância deve ser introduzida em blocos de dados durante o processo de codificação. Na fase de descompactação, cada um dos blocos deve ser processado com a intenção de se detectar e corrigir a ocorrência de erros. A Figura 5.19 apresenta o efeito da codificação em blocos considerando um comprimento de erro de 48 bits introduzidos ao longo do arquivo compactado. Apesar da utilização de um esquema de proteção para se limitar o efeito causado pelo erro de canal, o resultado obtido não se demonstrou satisfatório quando comparado ao da Figura 5.20.

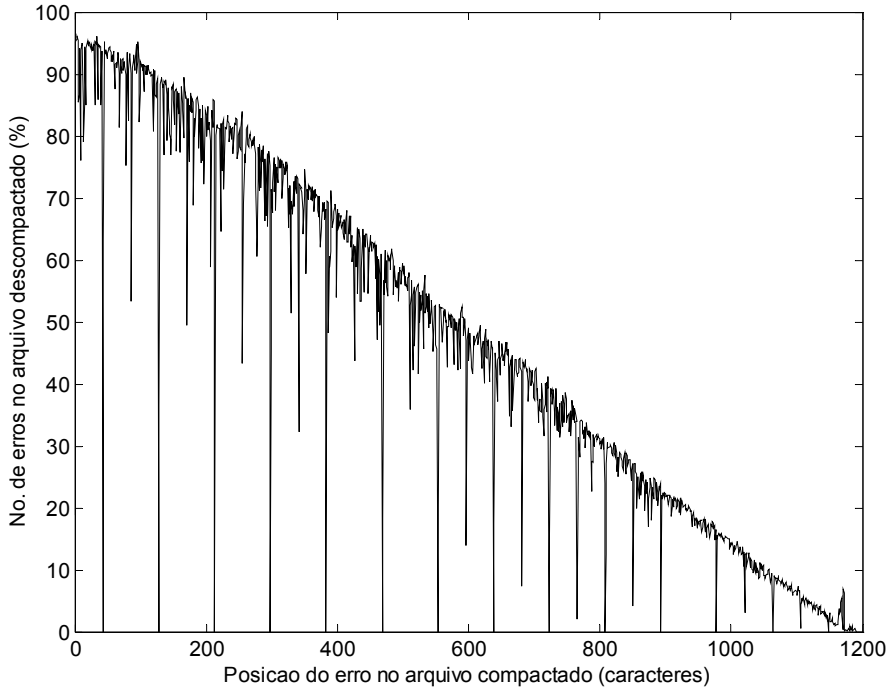

Fig 5.19. Impacto dos surtos de erro no arquivo compactado utilizando proteção uniforme contra os erros

5.3.2 Proteção Desigual Contra Erros no Arquivo Compactado

Para a Figura 5.20, o emprego da codificação RS unicamente sobre a seqüência de bits de flag, considerando surtos de erro de 48 bits com 12 bits de match e 16 bits de dicionário ao longo do arquivo compactado, demonstrou-se mais eficaz do que o método anterior. Os parâmetros do código de bloco linear Reed Solomon encontram-se descritos no capítulo 4. Comparando-se os dois últimos resultados, observou-se um valor médio de 6016 erros contra 956 erros propagados ao longo do arquivo durante o processo de descompactação, Figuras 5.19 e 5.20 respectivamente.

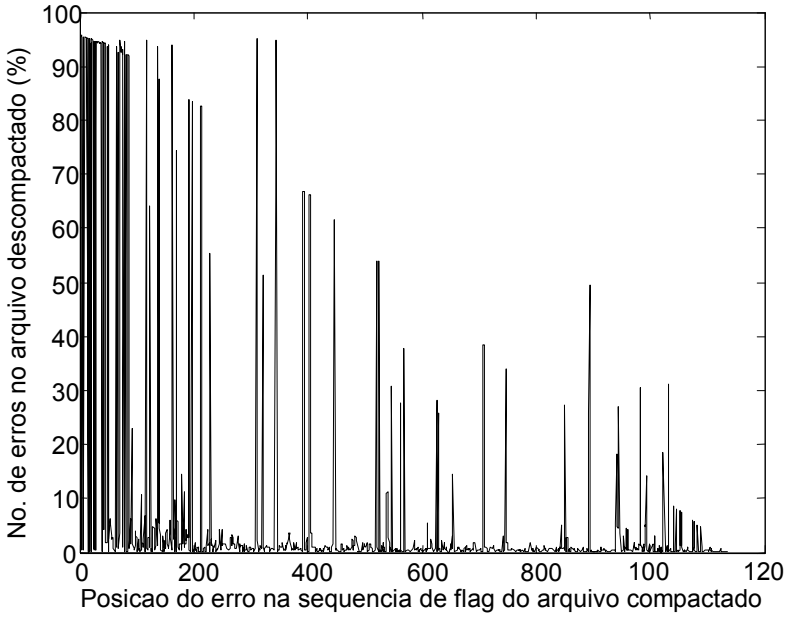

Fig. 5.20. Impacto dos surtos de erro no arquivo compactado utilizando proteção desigual de erros

5.4 Conclusão

Neste capítulo apresentamos os resultados encontrados nos testes realizados conforme descrito na seção 5.1. Os detalhes da análise a partir dos resultados apresentados poderão ser vistos a seguir.

Conclusões

O surgimento e o acesso a serviços móveis cada vez mais sofisticados e mais diversificados demandam um maior aproveitamento da largura de banda e uma maior otimização de espaço em disco ou de memória. Na transmissão de dados sem fio os erros são mais freqüentes e o tempo necessário para a recuperação dos dados é maior, por isso a proposta do estudo de se vincular a compactação de dados à proteção da ocorrência de erros desses dados compactados.

A ocorrência de erros nos dados compactados impacta em erros na descompactação, principalmente no que se refere a aplicações voltadas à transmissão sem fio. Em virtude deste fato, torna-se relevante o estudo dos códigos de compressão usando a proteção desigual de erros. Para tanto, este trabalho foi dividido em duas etapas. A primeira etapa visou o estudo do impacto de erros no algoritmo de compactação LZSS, avaliando as partes da informação compactada mais sensíveis à propagação de erros. A segunda etapa consistiu em propor uma estratégia de proteção desigual de erros, de maneira a minimizar a propagação de erros no processo de descompactação.

Para a análise do impacto dos erros nos diferentes componentes do código LZSS e a posterior implementação de uma estratégia de proteção desigual de erros, utilizou-se o aplicativo MATLAB®. Este aplicativo possui recursos computacionais que permitiram desenvolver um software para decompor a estrutura do arquivo LZSS e avaliar o efeito do impacto de erros. A criação deste programa possibilitou analisar a estrutura do arquivo compactado e do efeito dos surtos de erro de cada um dos componentes do código de compressão LZSS, a seguir discriminados: bits de flag, bits de offset, bits de match e bits de caractere, identificando onde o efeito dos surtos de erros provocou maior propagação.

A análise dos resultados do efeito de propagação destes erros durante o processo de descompactação do arquivo, após a inserção de surtos de erro em cada um dos componentes, demonstrou que o componente mais sensível a erros foi o componente de flag. Os surtos de erro nos bits de flag foram introduzidos seqüencialmente, gerando a propagação do erro sobre os bits de offset, os bits de match e os bits de caractere. Os resultados obtidos evidenciaram

que a porcentagem de erro que se propaga sobre o componente de flag é muito maior que as porcentagens de erro propagadas nos demais componentes. O arquivo gerado após a descompactação possuía um tamanho maior do que o arquivo original. Por esta razão optou-se por inserir o código de detecção e correção de erros Reed-Solomon, reduzindo o efeito da propagação de erro sobre o arquivo compactado. A utilização deste código mostrou-se mais eficiente quando aplicado a cada componente do arquivo, ao invés de aplicado sobre todo o arquivo texto compactado, tendo um ganho muito inferior em relação a proteção desigual.

Como proposta futura de pesquisa, recomenda-se a investigação de técnicas digitais para a compressão de dados sobre surtos de erro erros de canais, especificamente para a transmissão sem fio, como redes de área local sem fio (WLAN – Wireless Local Area Networks), redes de celular e de sensores. Recomenda-se ainda, a exploração da forma estruturada do arquivo compactado pelo sistema de transmissão, especificamente pelo esquiema de modulação codificado, para melhoria do desempenho do sistema. Para sistemas empregando esquemas híbridos de controle de erros FEC/ARQ, a recuperação parcial e retransmissão de informações compactadas podem incrementar o “throughput” do sistema e reduzir o consumo de energia de dispositivos portáteis.

Referências Bibliográficas

[BEL90] BELL, T. et al. *The Large Calgary Corpus zip file* Disponível em: www.data_compression.info/Corpora/CalgaryCorpus/ Acessado em mai/2005

[BOY81] BOYARINOV, I.M. and KATSMAN, G.L., *Linear Unequal Error Protection Codes.* IEEE Transactions on Information Theory. Vol. IT-27, no. 2, p168-175. Mar 1981

[COM04] Reed-Solomon Coding Tutorial 4i2i Communications Ltd 2004. Disponível em:http://www.4i2i.com/reed_solomon_codes.htm Acessado em mai/2005

[CAI96] G. CAIRE and G. LECHNER. *Turbo Code with Unequal Error Protection*. Electronics Letters. vol. 32, no. 7, Mar 1996

[FUJ00] FUJIWARA EIJI, CHEN HONGYUAN, KITAKAMI MASATO, *Error Recovery for Ziv-Lempel Coding by Using UEP Schemes*, ISIT 2000.

[GEL93] GELDREICH JR., R. *Programa Simple Hashing LZ77,. Sliding Dictonary Compression.* Disponível em: www.programmersheaven.com Acessado em mai/2005.

[HOF96] HOFFMAN ROY, *Data Compression in Digital Systems*, New York, Chapman & Hall, 1996

[HEL96] HELD, GILBERT AND MARSHALL, R. THOMAS, *Data and Image Compression Tools and Techniques*, New York, 4th Editon, John Willey and Sons LTD, 1996

[KRA00] KRASHINSKY, R. *Efficient Web Browsing for Mobile Clients using HTTP Compression.* Disponível em www.citeser.ist.psu.edu Acessado em jun/2005

[LIN83] LIN, S. & COSTELLO JUNIOR, D.J. *Error Control Coding: Fundamentals and Applications.* New Jersey: Prentice Hall, 1983.

[MAS67] MASNICK, B & WOLF, J. *On Linear Unequal Error Protection Codes.* IEEE Transactions on Information Theory. Vol. IT-03, no. 4, p600-607. Oct 1967

[MOS98] MOSS, K.N. *Recovery from Dropout Errors in LZ77 Compressed Text*. Data Compression Conference, 1998. DCC '98. Proceedings 30 March-1 April 1998 Page(s):565

[NEL91] NELSON, M. *The Data Compression Book*. M&T Publishing, 1991.

[PEL04] PELLENZ, M. *Notas de aula. Mimeo.* Curitiba, 2004

[PRO95] PROAKIS, John G. *Digital communications*. 3rd ed. New York: McGraw-Hill, 1995. 928p.

[RAM03] RAMOS, F. *Fundamentos do MatLab*. Estágio de Docência – PUCPR

[SAY00] SAYOOD, KHALID, *Introduction To Data Compression*. 2ND Edition, Morgan Kaufmann Publishers, 2000.

[STO82] STORER, JAMES and SZYMANSKI, THOMAS. *Data compression via Textual Substitution.* Journal of the association for computing machinery. Vol. 29, no. 4, p. 928-951, Oct 1982.

[WON01] WONG, JACKY MAN-LEE, BAO, PAUL and NG, VINCENT TO-YEE. *Incremental mining of association patterns on compressed data*. IEEE *magazine* 2001. p. 441-446

[YUA96] YUANFU, HU and XUNSEN, HU. *The methods of improving ratio of LZ77 family data compression algorithms*. Proceedings of ICSP' 96.

[ZIV77] ZIV, JACOB and LEMPEL, ABRAHAM. *A universal algorithm for sequencial data compression*. IEEE Transactions on Information Theory. Vol. IT-23, no. 3, p337-343.May 1977

[ZIV78] ZIV, JACOB and LEMPEL, ABRAHAM. *Compression of Individual Sequences via variable-rate coding*. IEEE Transactions on Information Theory. Vol. IT-24, no. 5, p530-536. Sep 1978