Pontifícia Universidade Católica do Rio de Janeiro
Departamento de Engenharia Mecânica
FAPERJ Fundação Carlos Chagas Filho de Amparo à Pesquisa do Estado do Rio de Janeiro

COMPRESSÃO DE DADOS SÍSMICOS

Aluno: Gabriel Cunha Manhães Pessanha

Orientadores: Marcelo Dreux
Maurício Meinicke

1. Introdução

Devido ao desenvolvimento das técnicas de aquisição de dados da subsuperfície terrestre e a sua importância para a indústria petrolífera, o volume de dados desta natureza tem crescido rapidamente. Este volume de dados tem gerado um problema para as empresas do ramo petrolífero, pois o volume de dados gerado na aquisição de uma bacia hidrográfica pode chegar à ordem de centenas de *terabytes*.

Outro problema é o uso de banda para o tráfego deste volume de informação pela rede corporativa. Quanto maior o volume de informação a ser trafegado, maior é a necessidade de banda. Muitas empresas possuem um servidor de dados, que são responsáveis por armazenar estes dados, em instalações que ficam fora da "sede" da empresa. Em função da segurança da informação, é desejado que estes dados sejam transferidos o mais rápido possível, para evitar que os mesmos sejam obtidos por pessoas indesejadas.

O dado sísmico possui características oscilatórias que dificultam sua compressão e a utilização de métodos com perda de informação pode comprometer processamentos futuros, gerando resultados equivocados.

Estão sendo estudadas alternativas que possibilitem a diminuição da quantidade de espaço em disco ocupado por estes dados, que poderão permitir a redução em custos de transporte e armazenamento dos mesmos.

O objetivo final do projeto é desenvolver uma técnica eficiente para a compressão dos dados sísmicos. Ao analisar a bibliografia existente sobre compressão de dados sísmicos, é observado que as técnicas de compressão de imagens podem ser aplicadas aos dados sísmicos com pequenas modificações. Pretende-se fazer um estudo sobre compressão de imagens, para em seguida aplicar estas técnicas ao dado sísmico.

2. Revisão Bibliográfica

2.1 Run-Length Encoding

Método amplamente utilizado e conhecido no campo da compressão de dados, o *Run-Length Encoding* (RLE) [1] é um tipo de codificação simples, mas que pode ser muito eficiente em alguns casos. O princípio básico por traz do RLE é representar uma sequência de símbolos repetidos por um contador e o próprio símbolo. Existem, no entanto, algumas variações no método para adaptá-lo a diferentes situações.

Criado a princípio como um eficiente método para compressão em máquinas de fax, onde o texto consiste em um conjunto de espaços em branco seguido de uma

pequena interrupção em preto. Devido a esses espaços, consegue-se reduzir bastante o tamanho do arquivo.

Considerando os caracteres a seguir:

A A A A A A B A A A B C C C C C C C B B B B C **(2.1.1)**

Para codificá-los de acordo com o método padrão, basta contar os caracteres que se repetem e representá-los pelo número de caracteres repetidos e pelo caractere em si, resultando em:

6 A 1 B 3 A 1 B 7 C 4 B 1 C

Neste exemplo, o método apresentou bons resultados na compressão. De 23 caracteres, passou a 14, apresentando uma redução de quase 40%. Essa redução substancial foi possível devido à presença de uma quantidade razoável de seqüências de símbolos repetidos. Pode-se notar, no entanto, que quando a mensagem não possui caracteres repetidos em sequência, sua representação dobra de tamanho, como nos casos 1 B, 1 B e 1 C. Com isso, quando há uma grande quantidade de símbolos que não se repetem sequencialmente, o desempenho de compressão do RLE pode vir a ser bem ineficiente, podendo até aumentar o número de símbolos necessários para representar o dado original.

As sequências abaixo mostram um caso onde o uso do RLE pode aumentar o tamanho do dado original, ainda que existam algumas séries de repetições.

Dado original: A B C A B C A A A A A A B C C C C C A B A A B **(2.1.2)**

Dado codificado: 1 A 1 B 1 C 1 A 1 B 1 C 6 A 1 B 5 C 1 A 1 B 2 A 1 B

Neste caso, depois de codificado, o dado passou a ter 26 símbolos, enquanto o original possui apenas 23 símbolos. No pior dos casos, onde o dado não possui nenhum símbolo repetido em sequência, ele irá dobrar de tamanho.

Uma alternativa ao método padrão, que pretende reduzir o problema para os casos onde há poucas repetições, é o RLE com *flag*. Nesta variação, quando o símbolo possui apenas uma repetição, o número 1 não é representado. Porém, não se pode simplesmente eliminar o número 1 e manter a mesma configuração anterior, pois a sequência ficaria ambígua. Fazendo a codificação da mensagem (2.1.2) utilizando o RLE com *flag*, tem-se

A B C A B C 6 A B 5 C A B 2 A B

O problema dessa representação é o fato de não se poder afirmar se o 6 e o 5 são realmente o número de repetições ou se são apenas símbolos comuns, pois é possível também aplicar o RLE a números.

A solução adotada foi a utilização de um *flag*, uma marcação para indicar que os próximos dois caracteres estão codificados e, ao decodificar a mensagem, estes devem ser expandidos. Neste caso, quando há uma série de apenas dois símbolos repetidos, eles não devem ser codificados, pois o resultado seria maior que o original. A utilização do símbolo de controle * para codificar os exemplos anteriores resulta em:

Mensagem (2.1.1): * 6 A B * 3 A B * 7 C * 4 B C
Mensagem (2.1.2): A B C A B C * 6 A B * 5 C A B A A B

Gerando seqüências de 15 e 18 caracteres respectivamente.

2.2 Codificação de Golomb

Com o contexto de transmitir uma série de resultados de um jogo de azar onde cada jogo consiste em uma seqüência de eventos favoráveis de mesma probabilidade até a primeira ocorrência de um evento não favorável, Solomon W. Golomb [2] desenvolveu uma codificação binária baseada no conceito da distribuição geométrica onde cada código tem comprimento variável.

Em seu trabalho, Golomb define p como sendo a probabilidade de ocorrência de um caso favorável e o parâmetro inteiro m tal que p^m=1/2. O número inteiro n, que representa a quantidade de casos favoráveis até a primeira falha, que é o que se quer codificar, possui probabilidade $G(n) = p^n q$. Para codificar da melhor maneira uma sequência de n números inteiros, Golomb chegou a conclusão que deveria haver m códigos de cada comprimento, exceto pelos comprimentos que são pequenos demais para ser representados sem ambiguidade e possivelmente alguns bits de transição que são usados menos de m vezes. Com isso para cada m é possível obter um dicionário de códigos. Este dicionário é uma representação de uma tabela de conversão entre o número inteiro a ser codificado e a sua representação binária. Através deste, também é utilizado para fazer a decodificação do número. Alguns exemplos demonstrados em seu trabalho:

	$m = 1$			$m = 2$	
n	$G(n)$	Codeword	n	$G(n)$	Codeword
0	1/2	0	0	0.293	00
1	1/4	10	1	0.207	01
2	1/8	110	2	0.116	100
3	1/16	1110	3	0.104	101
4	1/32	11110	4	0.073	1100
5	1/64	111110	5	0.051	1101
6	1/128	1111110	6	0.036	11100
7	1/256	11111110	7	0.025	11101
8	1/512	111111110	8	0.018	111100
9	1/1024	1111111110	9	0.013	111101
10	1/2048	11111111110	10	0.009	1111100

	$m = 3$			$m = 4$	
n	$G(n)$	Codeword	n	$G(n)$	Codeword
0	0.206	00	0	0.151	000
1	0.164	010	1	0.128	001
2	0.130	011	2	0.109	010
3	0.103	100	3	0.092	011
4	0.081	1010	4	0.078	1000
5	0.064	1011	5	0.066	1001
6	0.051	1100	6	0.056	1010
7	0.041	11010	7	0.048	1011
8	0.032	11011	8	0.040	11000
9	0.026	11100	9	0.034	11001
10	0.021	111010	10	0.029	11010

Figura 1. Tabela de dicionários do algoritmo de Golomg. Adaptado de [2].

Seja k o menor valor inteiro positivo tal que $2^k \geq 2m$, então o dicionário correspondente possui exatamente m códigos de comprimento maior ou igual a k e $2^{k-1} - m$ de comprimento igual a k -1. Quando o m for potência de dois, cairá em um caso mais simples onde não haverá códigos de comprimento $k - 1$.

Com isso, para construir o dicionário para um m que seja potência de dois, basta dividir o número a ser codificado pelo m. O resultado da divisão será o número de 1 que

aparecerão no início do código. Um zero indica o término da sequência de 1 proveniente da divisão, e o resto, desta divisão, é representado pela forma binária convencional com o número de algarismos necessário para representar o maior resto possível na forma binária, ou seja, o número necessário para representar o número *m-1*. Quanto menor o *m*, maior seu poder de compressão de números muito pequenos, porém sua representação para grandes números é muito maior.

Por não necessitar de informações além do *m* para a decodificação, ele pode apresentar resultados melhores em dados pequenos em comparação a métodos que precisem armazenar os dicionários. Além disso, tem resultados excelentes se a situação for similar à proposta pelo autor, com uma repetição de casos com probabilidade igual. Também pode ser eficiente para valores predominantemente baixos, mesmo que as probabilidades não sejam iguais.

Pelo fato da tabela de dicionários levar em consideração apenas o parâmetro *m*, e não a probabilidade de ocorrência de um determinado evento, o algoritmo de Golomb pode ser usado para a compressão de mensagens quando estas não são conhecidas totalmente. Ao definir a tabela de dicionários, a mensagens pode ser codificada de acordo com a demanda.

2.3 Codificação de Huffman

O método criado por David A. Huffman[3] representa cada símbolo por uma seqüência binária de acordo com a probabilidade de sua ocorrência. Diferentemente do método de Golomb, o código de Huffman requer que se tenha disponível toda a informação a ser codificada.

Ao utilizar a probabilidade de ocorrência de um determinado símbolo para definir a sua codificação, Huffman garante que os símbolos que possuem uma maior frequência sejam codificados com uma menor quantidade de bits que um símbolo que possua uma menor frequência.

Para cada símbolo, deve-se calcular a probabilidade de ocorrência de um determinado símbolo da mensagem, para depois montar uma árvore binária.

Para a criação da árvore, deve-se ordenar todos os símbolos presentes na mensagem de acordo com a sua probabilidade de ocorrência, que é dado pela divisão do número de ocorrências do símbolo pelo número total de símbolos. Os dois elementos de menor probabilidade são então adicionados como dois nós na árvore, sendo criado um nó auxiliar como pai, que não possui símbolo a ser comprimido, mas que possui probabilidade igual à soma das probabilidades das duas folhas que foram utilizados para a sua criação. Este novo nó deve adicionado aos símbolos da mensagem e deve-se reordenar os símbolos de acordo com a probabilidade de ocorrência. O processo é repetido até que a soma de probabilidades seja 1.

De acordo com essa árvore é então criado um dicionário que permitirá codificar e decodificar cada elemento. No entanto, como para cada arquivo as ocorrências e os símbolos serão diferentes, é preciso incluir o dicionário no arquivo comprimido.

O código de cada elemento é então construído de baixo para cima, começando da raiz até as folhas, onde cada folha terá armazenado um símbolo. Percorre-se cada nó, atribuindo 0 se o elemento estiver de um lado do nó ou 1 se estiver do outro.

Dependendo do modo de ordenação, pode haver diferentes códigos para um mesmo número de símbolos, como por exemplo, quando houver mais de um símbolo com o mesmo número de ocorrências.

Símbolo	1	2	3	4	5	6	8
Probabilidade	0,20	0,10	0,15	0,05	0,20	0,10	0,20

Tabela 1. Probabilidade de ocorrência de cada símbolo da mensagem.

Considerando a tabela acima, os símbolos podem ser ordenados em função de suas probabilidades da seguinte forma:{1 , 5 , 8 , 3 , 2 , 6 , 4}, gerando a árvore abaixo:

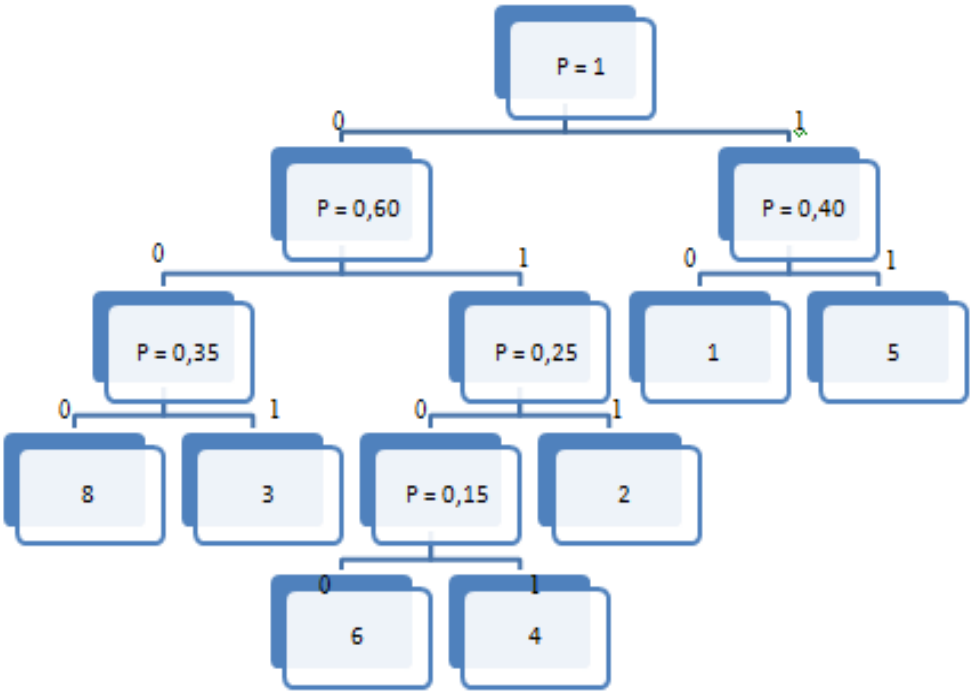

Figura 2. Representação da árvore utilizando os símbolos da tabela 1.

Resultando em:

Símbolo	1	2	3	4	5	6	8
Codificação	10	011	001	0101	11	0100	000

Tabela 2. Dicionário de Huffman para a mensagem a cima.

No entanto, esta não é a única codificação válida, pois outra ordenação possível seria {8 , 5 , 1 , 3 , 6 , 2 , 4}, assim obtendo:

Símbolo	1	2	3	4	5	6	8
Codificação	000	0100	001	0101	11	011	10

Tabela 3. Outra possibilidade para o dicionário de Huffman.

Entretanto, o tamanho final da mensagem será o mesmo. Deve-se ressaltar a importância do armazenamento correto dos dados da codificação, para evitar erros na decodificação.

O algoritmo obterá boas taxas de compressão quando o arquivo possuir uma grande repetição de símbolos, mas diferentemente do RLE eles não necessariamente precisam estar em ordem, fazendo-o uma poderosa ferramenta de compressão, principalmente em dados grandes, onde o tamanho da informação que deve ser armazenada para sua reconstrução não seja muito significativo. Justifica-se então seu uso nas mais diversas aplicações em compressão, apesar de originalmente ter sido pensado para textos em inglês.

2.4 Transformada *Wavelet* de Haar

Wavelets são funções matemáticas utilizadas para a decomposição e análise de funções complexas em funções mais simples. A transformada *wavelet* é muito utilizada na área de processamento de sinais. Normalmente comparada à transformada de Fourier, a transformada *wavelet* possui a vantagem de manter a correlação entre o espaço da frequência e o espaço do tempo.

Proposta em 1909 por Alfred Haar[4][5], muito antes de o termo *wavelet* ser definido formalmente, esta *wavelet* é considerada a mais simples. A base da transformada de Haar consiste em fazer uma série de médias e diferenças. Entretanto, este capítulo não se propõe a explicar e definir matematicamente em detalhes sua teoria, pois tornaria o texto demasiado longo e fugiria à sua proposta.

Considere um sinal discreto representado por { 7 , 1 , 3 , 5 }. É possível decompor este sinal, utilizando a transformada de Haar, ao computar a média entre os coeficientes, dois a dois, para conseguir uma representação deste sinal com menor resolução, que resulta em: { 4 , 4 }. Apenas com essa informação, no entanto, é impossível reconstruir sinal original. Para isso, são computados *coeficientes de detalhe*. Fazendo a diferença entre o primeiro coeficiente original do par e a média, obtém-se { 3 , -1 }. Agora, toda a informação necessária para a reconstrução do sinal existe, sendo esta representada por { 4 , 4 , 3 , -1 }. É importante observar que a transformada de Haar faz uma reordenação dos coeficientes que representam o sinal, pois é graças a esta reordenação que é possível ter as multiresoluções do sinal.

Para reconstruir o sinal original, basta somar o coeficiente de baixa resolução ao coeficiente de detalhe correspondente para obter o primeiro coeficiente do sinal original e subtrair para conseguir o segundo coeficiente. No exemplo anterior, para reconstruir o sinal original deve-se utilizar o primeiro coeficiente de baixa resolução, 4, junto com o primeiro coeficiente de detalhe, 3, para gerar os dois primeiros coeficientes do sinal original. O mesmo vale para os outros coeficientes. Sendo assim, o sinal original reconstruído fica: (4 + 3 = 7, 4 – 3 = 1, 4 + (-1) = 3, 4 – (- 1) = 5). O que mostra que a transformada *wavelet* de Haar é capaz de decompor e reconstruir um sinal sem acrescentar ruído ao mesmo.

O processo pode ser repetido para a parte das médias, resultando em { 4 }, com o coeficiente de detalhe { 0 }. No final, a série pode ser representada por { 4 , 0 , 3 , -1 }. Repare que o tamanho final da série é o mesmo da original, porém, o módulo de cada coeficiente será menor, e isso pode ser útil para a compressão.

A transformada *wavelet* de Haar possui uma limitação que é dimensão do sinal a ser decomposto. Por fazer médias dois a dois e por definir níveis de resolução, a transformada necessita que a dimensão, do sinal a ser decomposto, seja uma potência de dois.

Matematicamente, a *wavelet* de Haar é representada por dois tipos de funções bases. Uma é a função escala e a outra é a função de translação. Estas funções são definidas em um intervalo [0,1), e sua escala e translação é definidas de acordo com a resolução.

Pode-se imaginar a imagem como sendo um sinal discreto, cujos valores de cada pixel em uma linha, ou coluna, representam o valor do sinal em um determinado instante de tempo. Sendo assim, cada coeficiente da imagem corresponde a uma função constante, que pode ser representado por deslocamentos e variações no espaço vetorial da seguinte função:

$$\phi(x) \coloneqq \begin{cases} 1\ , & 0 \le x < 1 \\ 0\ , & fora \end{cases}$$

A função $\phi(x)$ é a representação da função escala descrita a cima para o nível de resolução original. A cada decomposição aplicada à imagem, o nível de resolução diminui e o limite do intervalo que a função $\phi(x)$ atua é reduzido à metade.

Os coeficientes de detalhe são definidos por outra função, a função de translação. Esta função é representada por:

$$\psi(x) \coloneqq \begin{cases} 1\ , & 0 \le x < \frac{1}{2} \\ -1\ , & \frac{1}{2} \le x < 1 \\ 0\ , & fora \end{cases}$$

Os limites de atuação da função $\psi(x)$ também são definidos de acordo com o nível de resolução que a imagem esta sendo decomposta.

Em *wavelets*, o conjunto de funções $\psi(x)$ e $\phi(x)$ que representam o conjunto de funções a serem aplicadas em um determinado nível de resolução é definido como sendo a *wavelet-mãe*.

2.5 Representação computacional de imagens

Um método muito comum de armazenamento de imagens em computação é por meio de uma matriz onde cada pixel é representado por uma cor. O tipo de representação de cor mais utilizado é o RGB, formado por três canais de cores, que são o vermelho, o verde e o azul, que ao serem combinados formam a cor desejada. Existem outros tipos de representação, como o CMYK que é utilizado pelos principais fabricantes de impressora. Neste padrão é utilizado um canal para o preto, outro para o ciano, o magenta, o amarelo e o preto. Neste trabalho foram utilizadas apenas as imagens no sistema RGB.

As imagens estudadas no projeto, e que representam parte significativa na utilização em computadores, utilizam 1 byte para representar a intensidade de cada canal, podendo assim gerar 256 variações, onde o 0 representa a ausência daquela cor e o 255 seu valor máximo. Com esta representação, o número total de cores, que podem ser representadas, é dado por 16.777.216. Este número representa o padrão de cores *TrueColor* utilizado pelo *Windows*.

3. Metodologia

Em geral, os trabalhos de compressão de dados volumétricos encontrados, sejam dados sísmicos, imagens médicas e outros, utilizam as técnicas de compressão de imagens ou alguma variação destas técnicas. Por isso, houve a necessidade de realizar um estudo prévio sobre compressão de imagens. Os dados de imagem possuem grande quantidade de informações, úteis para comparação e validação dos métodos utilizados.

Para a execução dos testes, foi desenvolvido um programa que possibilita a compressão e a descompressão de imagens, utilizando diferentes algoritmos de compressão, variando a ordem de aplicação e parâmetros destes algoritmos. Atualmente o programa é capaz de utilizar os códigos RLE, na versão padrão ou com flag, e

Golomb para a compressão e descompressão de imagens. O algoritmo de Huffman está em estágio final de implementação e será o próximo a ser incluído no programa.

A interface gráfica foi desenvolvida utilizando a biblioteca IUP[6], criada pelo laboratório Tecgraf/Puc-Rio, por ser de uso livre para o meio acadêmico e por estar disponível para o sistema *Windows* e *Linux*. O desenho das imagens é feito utilizando a biblioteca *OpenGL*.

A interface é composta por duas telas para carregamento das imagens, uma aba de controle dos parâmetros para a compressão da imagem e uma aba para a descompressão de um arquivo. Ao carregar uma imagem, ela irá ser exibida na tela da direita, quando então pode ser comprimida utilizando a aba de controle. Ao descomprimir uma imagem ela aparece na tela da direita.

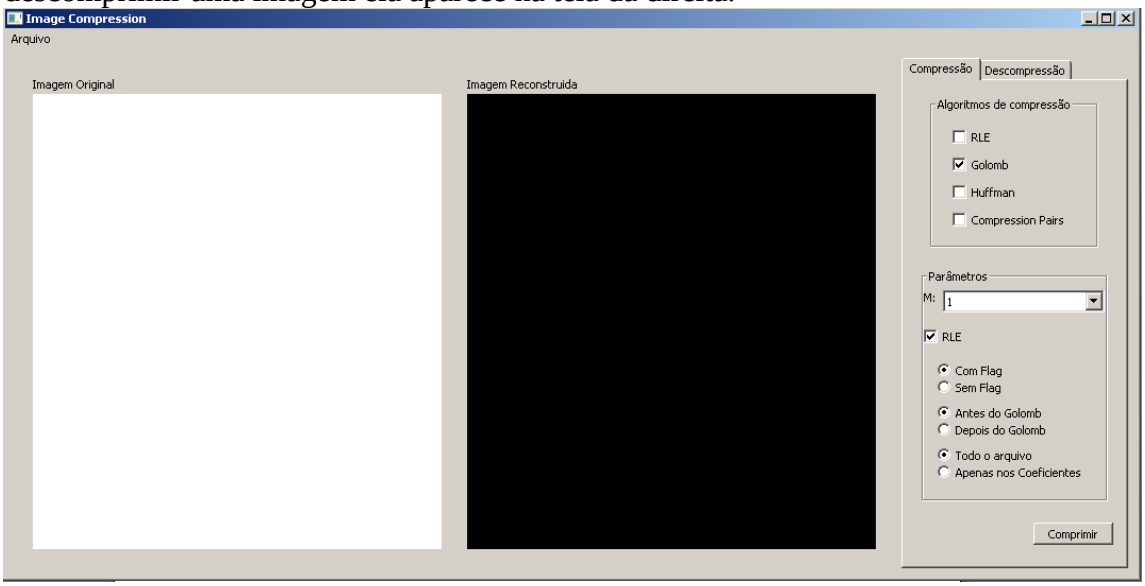

Figura 3. Software Image Compression desenvolvido para testar os algoritmos de compressão de imagens.

Decidiu-se por implementar os códigos de compressão a partir de seus artigos originais ou referências puramente teóricas, sem a consulta a códigos prontos. Os códigos foram criados na linguagem C++ e escritos de forma modular, para que possam ser aproveitados em outros programas.

3.1 Run-Length Encoding

Primeiro algoritmo a ser desenvolvido no projeto, o RLE padrão possui uma implementação simples, onde bastam basicamente duas variáveis para comparar os pixels em seqüência e um contador para quantidade de símbolos repetidos para reescrevê-los. Para armazenar o contador, foi escolhido um tipo de variável de 1 byte, mesmo tamanho de cada símbolo utilizado nas imagens. A escolha é útil, pois é o menor valor possível de se manipular diretamente utilizando a linguagem escolhida, mas ainda assim apresenta um valor máximo alto o suficiente para grandes taxas de compressão se a imagem possuir muitas séries de repetições.

Para aproveitar melhor as repetições dos bytes, a imagem é armazenada na forma RRGGBB, ao invés da mais comum RGBRGB. Isso significa que cada canal é armazenado em sequência, e não com os canais intercalados, como normalmente é feito.

Como todos os valores possíveis em um byte são utilizados para representar a intensidade dos canais RGB, no caso do RLE com *flag*, não se pode atribuir um valor para o símbolo de controle que não seja ambíguo sem utilizar mais de 1 byte para representá-lo. A solução encontrada foi convencionar um número que faça parte dos símbolos permitidos para a imagem e quando ele for codificado, tratá-lo como se fosse uma seqüência mesmo que de tamanho 1. No algoritmo, foi utilizado o 2 como *flag*. Para exemplificar o que ocorre no programa, considere a codificação a seguir:

Dado original: 1 1 1 1 1 2 3 3 3 3 3 3 3 1 1 3 1 3 2 2 2 2 **(3.1.1)**

Dado codificado: 2 5 1 2 1 2 2 7 3 1 1 3 1 3 2 4 2

No dado original, ao aparecer o símbolo 2 este é representado no dado codificado como sendo a sequência { 2 1 2 } e { 2 4 2}. É evidente que quando não há repetição ou esta é menor que três, o algoritmo irá aumentar a quantidade de símbolos usados para representar esta corrida.

3.2 Codificação Golomb

A implementação do Golomb foi tratada de duas maneiras distintas. A primeira consiste em aplicar o RLE em toda a imagem e depois aplicar a codificação do Golomb apenas nos números de repetição de cada byte. A segunda foi aplicar a codificação em toda a imagem. Em ambos os casos, no entanto, não é nada comum que a ocorrência dos números respeite a distribuição geométrica e tenham sempre as mesmas probabilidades, assim como no exemplo utilizado para a criação do código, por isso foram testados diferentes valores de *m*.

Por questões de simplificação na construção dos dicionários, foram utilizados apenas valores de *m* que são potência de 2, restringindo ao caso onde sempre há *m* códigos de cada tamanho. Seu tamanho máximo foi restrito a 128, por se tratar da maior potência de 2 possível de ser representada por 1 byte, considerando que haja o 0.

Escolhido o parâmetro *m* uma função constrói o dicionário correspondente. Assim percorre-se o arquivo substituindo os valores por sua codificação em uma só passada. Por se tratar de um código binário, são necessárias algumas conversões, pois a menor unidade tratada diretamente pela linguagem C++ é o byte, e o método é muito mais eficiente em termos de compressão com a utilização do bit.

Para a decodificação, o arquivo salvo é convertido de volta para a representação binária e ele é decodificado “bit a bit”, de acordo com a regra de criação do *m* utilizado.

3.3 Codificação Huffman

Para utilizar a codificação de Huffman, o algoritmo criado percorre o arquivo, mapeando cada símbolo e sua ocorrência e os armazenando em uma estrutura. Depois, é criada a árvore de acordo com tais ocorrências. Uma vez pronta, cria-se o dicionário de acordo com a posição na árvore de cada folha, que corresponde a cada símbolo. Finalmente, percorre-se novamente o arquivo para converter os elementos. O algoritmo já possui todas essas funções implementadas.

Para a decodificação, é necessário ter o dicionário que foi utilizado para a codificação. Este dicionário pode ser guardado junto com a mensagem codificada ou,

tendo o mapa de símbolos e ocorrências, pode-se recriar o dicionário levando em consideração os mesmos critérios utilizados para a ordenação dos símbolos que originou o dicionário.

Com o dicionário, percorre-se o arquivo desfazendo as codificações. A função de desfazer a codificação tendo o dicionário já está completa. A parte pendente do algoritmo está em salvar as informações necessárias para poder recuperar o dicionário posteriormente. Está sendo estudada a melhor forma de salvar o mapa ou o dicionário no arquivo e sua leitura ao abrir um arquivo já salvo.

3.4 Transformada Wavelet de Haar

Foi desenvolvido um programa para aplicar a wavelet de Haar em imagens. Sua interface, também criada com a biblioteca IUP apresenta duas telas para imagens e uma janela de controle. A Tela da direita mostra a imagem completa, com os coeficientes de imagem e os coeficientes de detalhe. A tela da esquerda mostra apenas os coeficientes de imagem, ampliada para que se possa notar os efeitos quando a resolução é diminuída.

Imagens são seqüências bi-dimensionais, e há duas maneiras comuns de utilizar a transformada para o caso de duas dimensões.

O modo padrão (Figura 4) consiste considerar cada linha como uma seqüência unidimensional e aplicar a transformada até chegar ao número máximo de transformadas. Depois, repete-se o mesmo processo, mas nas colunas da imagem.

O método não-padrão (Figura 5) consiste em aplicar alternadamente as transformadas, uma vez nas linhas e outra nas colunas. O método não-padrão foi escolhido por apresentar coeficientes de detalhes de melhor visualização e por ser mais veloz computacionalmente.

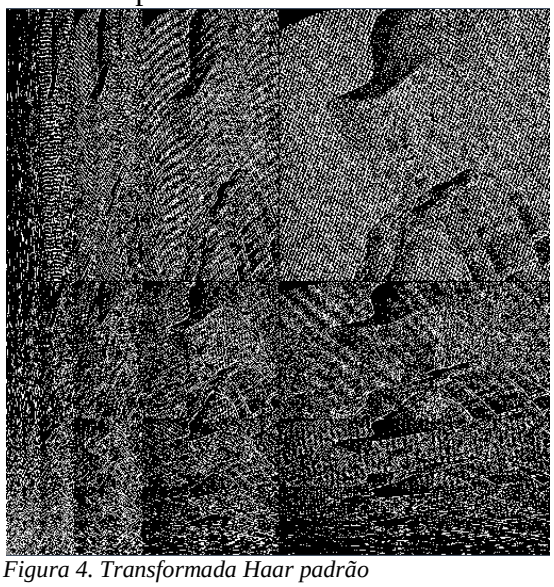

Figura 4. Transformada Haar padrão

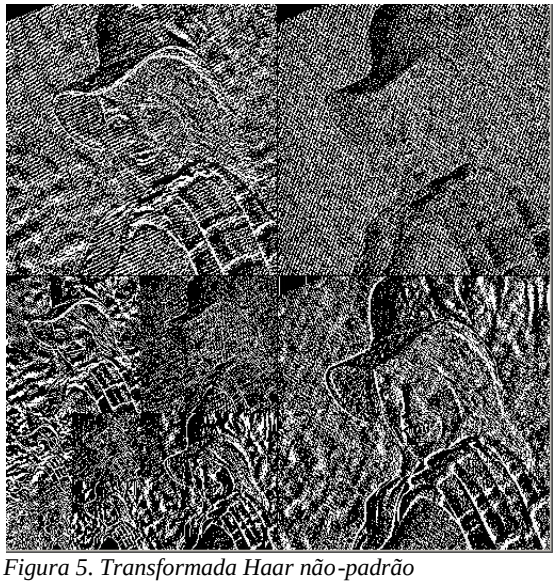

Figura 5. Transformada Haar não-padrão

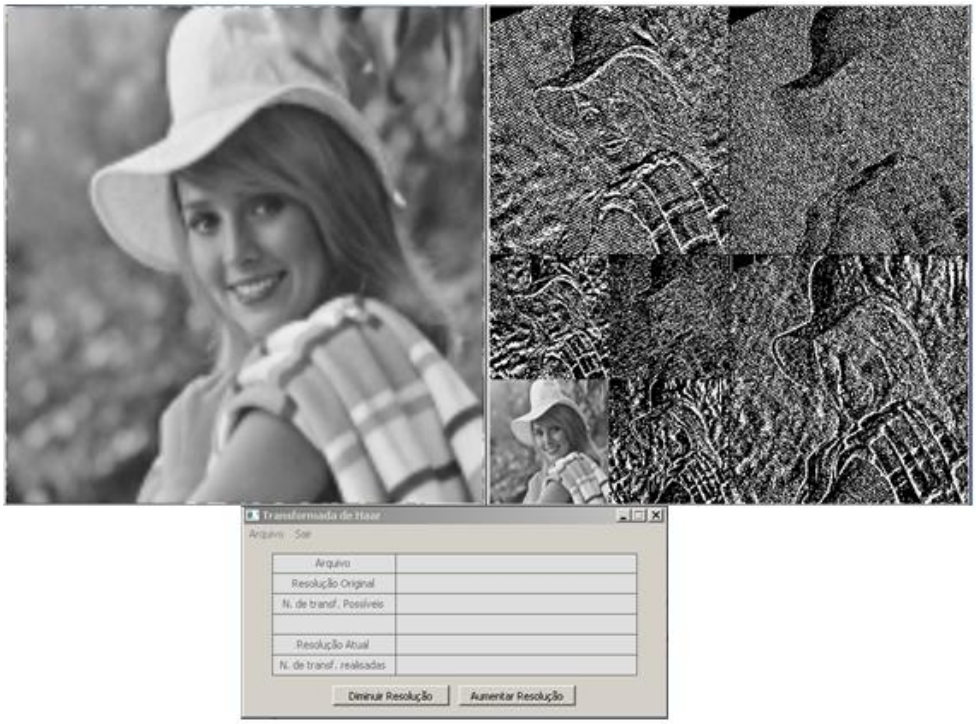

Figura 6. Software Transformada de Haar desenvolvido para auxiliar os estudos sobre a transformada wavelet de Haar.

4. Resultados

Os resultados abaixo representam a utilização dos algoritmos Golomb, RLE e RLE com *flag*, em três imagens do *USC-SIPI Image Database*, banco de dados de imagens da *University of Southern California* e uma imagem gerada utilizando um programa de computador. Outros testes foram feitos, mas as figuras abaixo representam os padrões de comportamento das compressões obtidos.

Cada curva no gráfico representa, no eixo *y*, o tamanho da imagem, em bytes, após a codificação por cada método ou combinação de métodos e, no eixo *x*, o valor do parâmetro *m* utilizado pelo código de Golomb. Quando o método do Golomb não é utilizado, o resultado é uma reta, pois o parâmetro *m* não faz sentido para os demais casos.

Legenda dos Gráficos:

Imagem Original
Código de Golomb
Código Rle Padrão
Código Rle com Flag
Código RLE padrão com a aplicação do código Golomb no número de repetições do RLE
Código RLE com flag com a aplicação do código Golomb no número de repetições do RLE
Código RLE com flag com a aplicação do código Golomb em toda a imagem

Tabela 4. Legenda para os gráficos que serão apresentados a seguir.

Figura 7. Imagem retirada do USC-SIPI Image Database paratestes de compressão

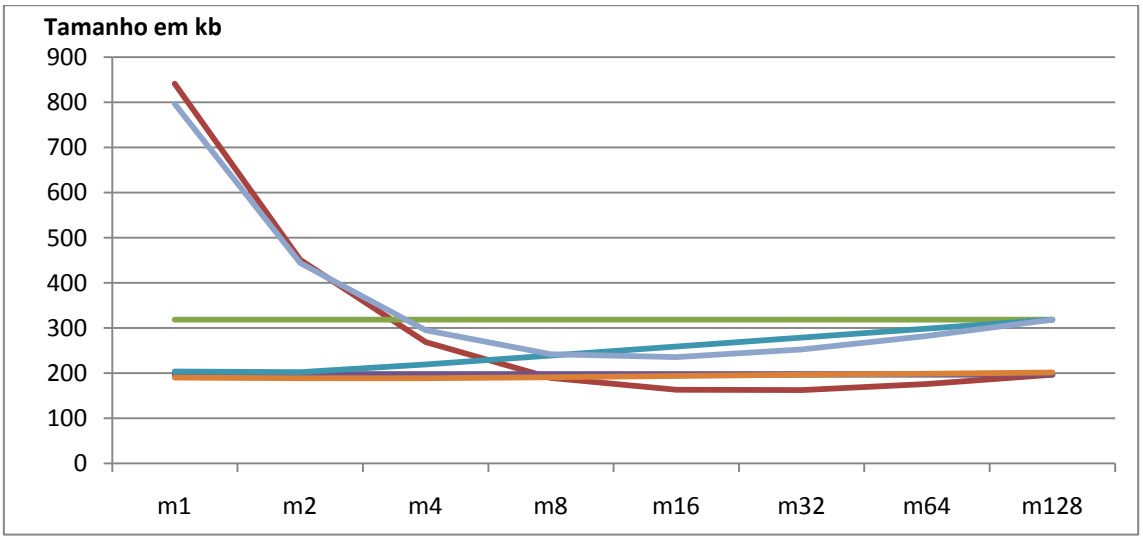

Gráfico 1. Resultados das codificações para a figura 7.

Para a figura 7, nota-se que os resultados do RLE padrão foram muito ruins em termos de compressão, aumentando o dado consideravelmente. Isso ocorreu porque há poucas repetições seguidas na imagem. Em geral, fotografias não são bem comprimidas pelo RLE padrão, pois a ocorrência de pixels repetidos é baixa. Por isso, a utilização do RLE com *flag* foi bem próxima do dado original, seja ele com ou sem o Golomb nos coeficientes.

Quando aplicado o Golomb nos coeficientes do RLE padrão os resultados foram piorando com o aumento do *m*, pois a maioria dos coeficientes era 1, e para este valor, o *m1* gasta apenas 1 bit, fazendo-o com que fique próximo ao original enquanto com o *m128* o número 1 passa a ter 8 bits, deixando-o muito próximo ao rle padrão.

Por ser uma imagem predominantemente escura, ou seja, os pixels são de valores predominantemente baixos, o Golomb em todo o arquivo se mostrou uma forma de compressão razoável, quando utilizados os valores 16, 32 e 64 para o *m*, embora com um ganho não tão grande em relação ao tamanho da imagem original. Para valores menores do *m*, os resultados foram ruins, e isso deverá ocorrer para a grande maioria das imagens que não sejam muito próximas ao preto, pois, tomando como exemplo o *m* = 1, quando o valor de sua cor é o 0, ele apresenta incríveis taxas de compressão de 1/8 do tamanho original (de 8 bits passa a 1 bit). No entanto, no caso do extremo oposto, 255, seu tamanho é multiplicado por 32. (de 8 bits passa a 256).

Figura 8. Imagem retirada do USC-SIPI Image Database para testes de compressão

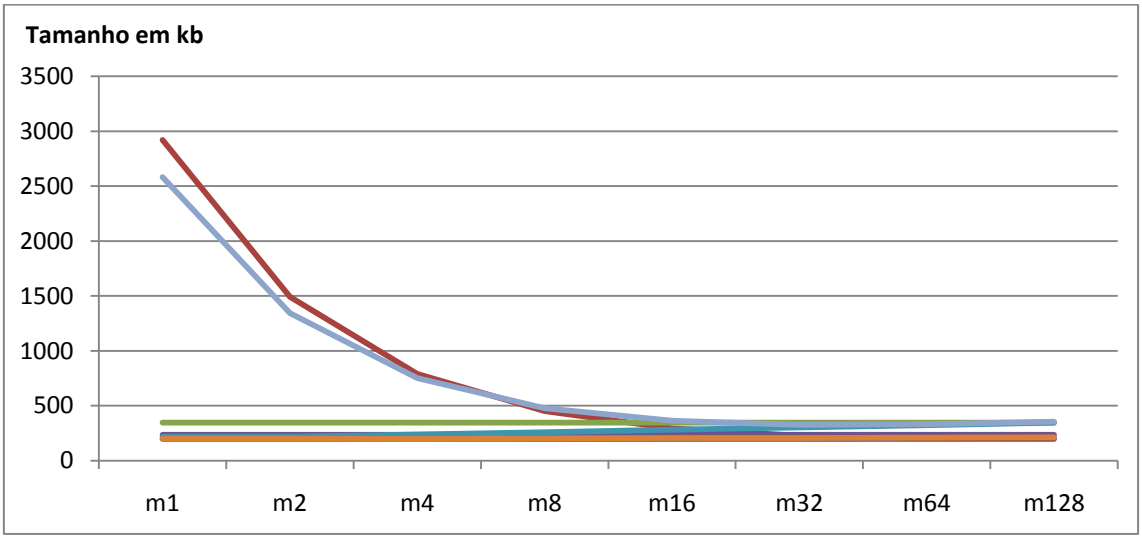

Gráfico 2. Resultados das codificações para a figura 8.

A figura 8 representa um padrão que não apresentou resultados favoráveis à compressão com praticamente nenhum dos algoritmos aplicados. É uma imagem clara e onde há poucas sequências de repetição. Nenhuma das combinações apresentou uma redução substancial no tamanho do arquivo, a única que não aumentou seu tamanho foi o RLE com *flag* nos coeficientes, utilizando um $m = 4$, mas reduziu em apenas 0,01% de seu tamanho. No entanto, espera-se que o algoritmo Huffman apresente resultados favoráveis.

Figura 9. Imagem gerada pelo autor para testes de compressão

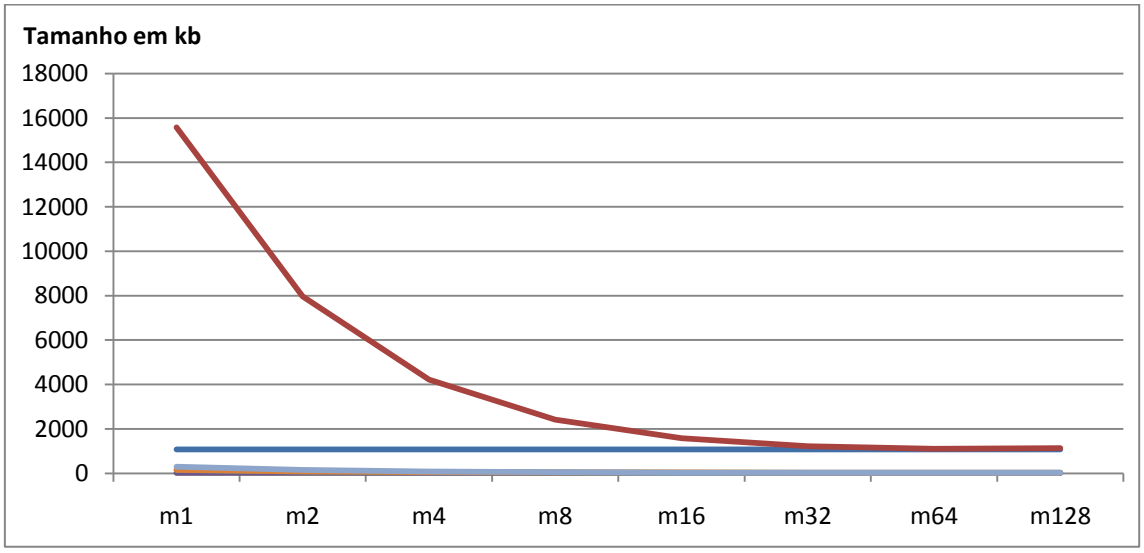

Gráfico 3. Resultados das codificações para a figura 9.

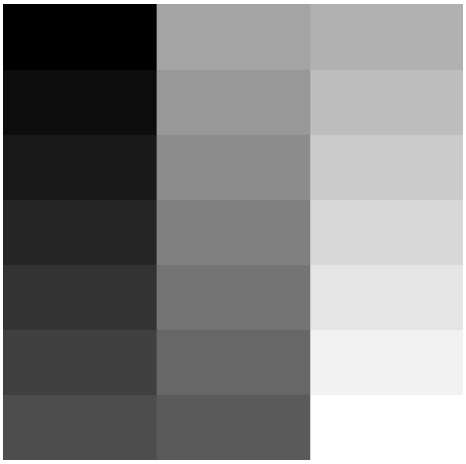

Figura 10. Imagem retirada do USC-SIPI Image Database para testes de compressão

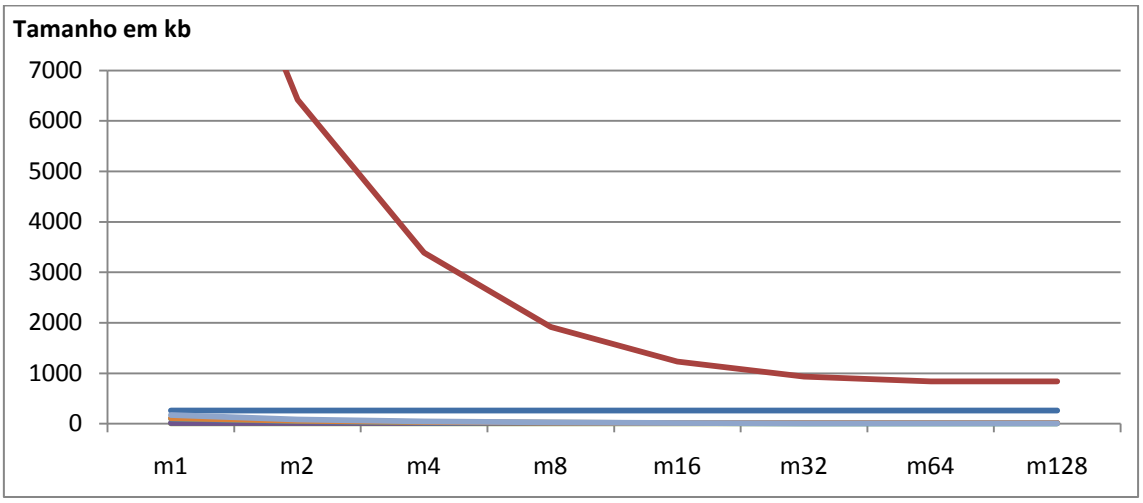

Gráfico 4. Resultados das codificações para a figura 10.

As figuras 9 e 10 apresentaram excelentes resultados em todos os testes que utilizam o RLE. No entanto, o melhor resultado foi utilizando o RLE padrão sem o Golomb. Isso ocorreu porque o arquivo apresenta grandes sequências de bytes repetidos. A escala da figura 10 foi editada para que fosse possível ver que a imagem original é maior que os casos onde o RLE é utilizado. Os valores quando há apenas o Golomb são muito grandes. Apesar de a figura apresentar As mesmas proporções entre claro e escuro no total de pixels, o Golomb faz com que a partes muito claras aumentem muito mais o tamanho se comparado a economia que fornece às partes escuras.

5. Conclusões e trabalhos futuros

Foram desenvolvidos algoritmos de compressão a partir de teorias bem consolidadas na literatura, o *Run-Length Encoding*, em seu formato padrão e em sua variação utilizando um *flag*, e o *Golomb*. Foram realizados testes com diversas imagens disponíveis em bancos de dados e outras pertencentes ao autor deste trabalho. Os resultados são compatíveis com o esperado teoricamente.

Foi desenvolvida a transformada *wavelet de Haar* para a aplicação em imagens. Espera-se obter resultados melhores ao combinar a transformada *wavelet* de Haar com os algoritmos de compressão. Serão feitos novos testes para validar essa teoria.

Atualmente, está em desenvolvimento o algoritmo de *Huffman*. Espera-se obter resultados melhores e ligeiramente menos dependentes do comportamento de cada imagem.

6. Referências

[1] Salomon, David; “Data Compression – The Complete Reference”; Springer Sciense + Business Media; ISBN-10: 1-84628-602-6; fourth edition, 2006.
[2] Golomb, S. W., “Run-length encodings” - (1966); IEEE Trans Info Theory 12(3):399
[3] Huffman, David; “A Method for the Construction of Minimum-Redundancy Codes”, Proceedings of the IRE, vol.40, no.9, pp.1098-1101, Sept. 1952.

[4] Haar, Alfred, "*Zur Theorie der orthogonalen Funkionensysteme"* [On the theory of orthogonal function systems], Mathematische Annalen, 69 (1910), 331-371. Translated by Georg Zimmermann.

[5] Stollnitz, Eric; DeRose, Tony; Salesin, David; *"Wavelets for Computer Graphics, Theory and Applications"*; Morgan Kaufmann Publisher, Inc.; ISBN 1-55860-375-1; 1996.

[6] Levy, C. H. "IUP/LED: Uma Ferramenta Portátil de Interface com Usuário". M.Sc. dissertation, Computer Science Department, PUC-Rio, 1993.