Projeto 1 - Código de Huffman

Lucas Campos Jorge
dept. de Engenharia Elétrica (ENE)
Universidade de Brasília (UNB)
Brasília, Brasil
lucascj29@gmail.com

I. Introdução

Codificação de Huffman é uma codificação sem perdas que possui o objetivo de reduzir a redundância e se aproximar da entropia do sinal. Nesta técnica de codificação são utilizados códigos de comprimento variável, onde serão a atribuídos aos símbolos mais frequentes, códigos menores e para os menos frequentes, códigos maiores.

O projeto desenvolvido tem o objetivo de implementar a codificação de Huffman e avaliar seus resultados em um conjunto de arquivos de diferentes fontes. Para a implementação foi escolhido a linguagem de programação Python.

II. Implementação

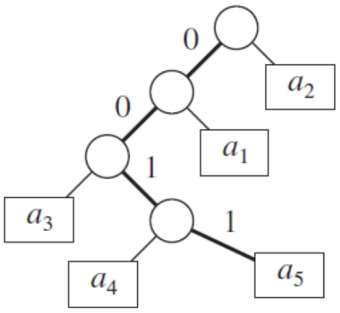
A. Algoritmo

O algoritmo se baseia na realização das seguintes etapas:

- Ordenar os símbolos em ordem decrescente de probabilidade, neste caso foi utilizada a frequência dos símbolos no arquivo de entrada.
- Para os símbolos menos frequentes são atribuídos os códigos $\alpha_i.0$ e $\alpha_i.1$. Onde α representa um vetor binário que será completado
- Agrupar os menos frequentes como um novo símbolo
- Repetir os passos acima até sobrar sobrar somente dois símbolos e atribuir 0 e 1 como códigos.

O resultado do procedimento descrito acima pode ser melhor visualizado no formato de uma árvore. Como na imagem abaixo:

Figura 1. Árvore de Huffman



1) Codificador: O codificador possui a função de comprimir o arquivo de entrada utilizando a codificação de Huffman, semelhante ao algoritmo descrito acima. Neste caso, o arquivo é considerado como uma fonte de símbolos de 8 bits, onde será lido byte a byte. Logo, após sua execução se obtém a cada símbolo presente no arquivo, um código.

Como saída, o codificador deve gerar um arquivo comprimido, onde este arquivo desse estar presente os códigos de cada símbolo e o conteúdo do arquivo de entrada com os símbolos substituídos pelos códigos calculados. A implementação utilizada envia no início do arquivo de saída as informações necessárias para a decodificação. Portanto o arquivo comprimido possui a seguinte estrutura.

- (1 byte) A quantidade de bits do maior código gerado.
- (3 bits) O tamanho do padding necessário para que o arquivo mantenha sua estrutura em bytes
- Para cada símbolos em ordem crescente ($s_0 = 00000000$ ... $s_{255} = 11111111$) um bit que determina se símbolo possui código (1 - possui código, 0 - não possui código) seguido do seu tamanho em bits (escritos com a quantidade de bits miníma para descrever o tamanho do maior código gerado) e por fim, o código.
- (0 - 7 bits) Padding
- Dados codificados

2) Decodificador: O decodificador possui a função de ler o arquivo comprimido e recuperar os dados originais antes da codificação. Neste caso, foi utilizado o método de lookup table, onde, após a leitura da informação lateral inicial do arquivo cria-se uma tabela que possui cada símbolo e código. Portanto, após a leitura de todos os códigos, basta ler os dados comprimidos, escrevendo no arquivo de saída cada símbolo referente a cada código.

B. Programa em Python

A implementação da codificação de Huffman foi realizada em Python. Onde se utilizou a biblioteca BitString para tornar possível a manipulação de um conjunto de bits como vetores.

Dentre as principais estruturas de dados, tem-se a classe Tree que possui os dados de cada símbolo e uma referência a mesma classe como direita e esquerda utilizadas para a criação da estrutura da árvore de Huffman.

III. Resultados

A duas aplicações desenvolvidas, o codificador e decodificador, foram executados em cima de 7 arquivos que possuem diferentes finalidades. Dentre os arquivos, um livro, uma imagem e códigos fonte. Para tais arquivos podemos comparar a entropia do sinal, o comprimento médio do código gerado, o tamanho (número de bits) da sequência codificada (com e

sem a informação lateral adicionada). Além disto, comparar com um codificador comercial, o Zip.

- Arquivos

Arquivos	***Tamanho (Bytes)***
dom_casmurro.txt	389670
fonte.txt	1000000
fonte0.txt	1000
fonte1.txt	1000000
lena.bmp	263290
TEncEntropy.txt	19415
TEncSearch.txt	253012

Tabela I: Tabela de arquivos

- Dados de codificação

Arquivos	***Entropia do sinal (Bits/Símbolo)***	***Comprimento médio (Bits/Símbolo)***
dom_casmurro.txt	4,643	4,685
fonte.txt	2,894	2,95
fonte0.txt	1,583	1,968
fonte1.txt	3,303	3,327
lena.bmp	7,51	7,545
TEncEntropy.txt	5,272	5,375
TEncSearch.txt	5,158	5,198

Tabela II: Tabela de dados da codificação

- Comparação de tamanho das compressões

Arquivos	***Tamanho s/ informação lateral (Bytes)***	***Tamanho total (Bytes)***	***Comercial Zip (Bytes)***
dom_casmurro.txt	227983,875	228217	147281
fonte.txt	268692,625	368735	428958
fonte0.txt	208,5	246	875
fonte1.txt	415792,5	415850	490348
lena.bmp	247809,375	248306	227779
TEncEntropy.txt	12881,5	13044	4793
TEncSearch.txt	164195,625	164387	39242

Tabela III: Tabela de tamanhos da codificação

IV. Conclusão

Podemos observar a utilidade e a eficácia da codificação de Huffman em diferentes arquivos com tamanhos e propósitos distintos.

Dentre os resultados relevantes, a compressão da imagem (lena.bmp) obteve a pior compressão dentre os arquivos, devido a sua alta entropia que limita a efetividade da codificação (sem perdas) de Huffman. Os melhores resultados, podem ser observados em arquivos de texto com entropia baixa, onde obtivemos pouca redundância na codificação, como no arquivo fonte1.txt. Entretanto, em arquivos muito pequenos é notável a influência da informação lateral na compressão, comparando a entropia do sinal e o comprimento médio do código, como no arquivo fonte0.txt.

Ao compararmos com um codificador sem perdas comercial, como o Zip, observamos que a codificação foi superior em aquivos pequenos e de baixa entropia, dentre eles, fonte0.txt e fonte.txt.

Diversas melhorias podem ser realizadas na compressão. Um dos pontos principais, onde já existem técnicas melhores, é o tamanho da informação lateral. Uma das técnicas existentes é codificação de Huffman canônica que tem o propósito de melhoria neste aspecto, pois permite que se envie somente o tamanho do código de cada símbolo. Além disto, podemos aumentar o tamanho dos símbolos, os tornando variáveis. Este tipo de abordagem pode impactar significativamente a codificação em arquivos de texto, onde há repetições de palavras, por exemplo.

Referências

[1] Slides de Aula - Disciplina de Fundamentos de Compressão de Sinais - Universidade de Brasília - Prof. Edson Mintsu Hung