BST and Heap : Huffman coding and decoding

Romão.H.L, Ribeiro. P.V.S, *Member, IEEE*

Resumo

Este projeto implementa o algoritmo de codificação e decodificação de Huffman, amplamente utilizado na compressão de dados. O sistema irá receber um arquivo HTML e construir uma árvore binária de Huffman, usando uma fila de prioridade para selecionar nós de menor frequência. Em seguida, será criado um dicionário de códigos, permitindo a conversão de frases em código de Huffman e sua posterior decodificação. A implementação inclui as estruturas de dados BST e Heap. Recursos externos como GraphViz foram empregados para a visualização gráfica da árvore de Huffman. Neste relatório, consta o método de desenvolvimento do algoritmo de codificação e decodificação de Huffman, bem como o seu funcionamento, análise dos algoritmos utilizados em sua implementação e alguns testes realizados.

***Palavras-chave: Algoritmo , Compressão, Dicionário, Huffman*.**

Abstract

This project implements the Huffman coding and decoding algorithm, widely used in data compression. The system receives an HTML file and constructs a Huffman binary tree using a priority queue to select nodes with lower frequencies. Subsequently, a code dictionary is created, enabling the conversion of phrases into Huffman code and their subsequent decoding. The implementation includes data structures such as BST and Heap. External resources like GraphViz were employed for graphical visualization of the Huffman tree. This report presents the development method of the Huffman coding and decoding algorithm, its functionality, analysis of the algorithms used in its implementation, and some conducted tests.

Keywords: Algorithm, Compression, Dictionary, Huffman.

I. Introdução

Nos últimos séculos, houve um enorme avanço tecnológico, resultando em um aumento significativo no volume de dados circulando na internet. Isso levou à necessidade de encontrar maneiras mais eficientes de transmitir dados entre computadores.

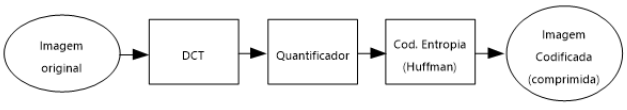
Nesse contexto, em 1952, David A. Huffman desenvolveu um dos algoritmos clássicos mais conhecidos para compressão e descompressão de dados, chamado de algoritmo de Huffman.

O algoritmo de Huffman é baseado na identificação das frequências de ocorrência de cada palavra no documento a ser compactado. Por exemplo, considerando a palavra *"abracadabra"*, podemos observar que o caractere *"a"* aparece 5 vezes, o *"b"* aparece duas vezes, o *"r"* também aparece duas vezes, e assim por diante. A regra é utilizar o menor número possível de bits para representar cada informação.

Durante a execução do algoritmo, uma árvore de Huffman é criada, contendo todos os caracteres presentes no documento. Essa árvore é utilizada para mapear cada caractere em uma sequência de bits equivalente.

Como mencionado anteriormente, a codificação de Huffman é amplamente utilizada em tecnologias multimídia. Esse algoritmo, que permite otimizar informações sem perdas, é amplamente empregado como etapa final no processamento de dados em normas como JPEG (usada para compressão de imagens), MPEG (utilizada em vídeos) e GZIP (compressão de arquivos).

Em geral, a codificação de Huffman é aplicada como o último bloco em uma sequência de vários blocos, de acordo com a norma. Por exemplo, ao codificar uma imagem usando a norma JPEG, os componentes de processamento podem ser esquematizados da seguinte maneira:



[1]Este trabalho é relativo ao projeto final da disciplina Análise de Algoritmos (DCC-606) do Professor Doutor Herbert Oliveira Rocha.

H. L. Romão, Aluno na Universidade Federal de Roraima (UFRR), discente do Curso de Ciências da Computação, do Departamento de Ciências da Computação (DCC), Boa Vista - Roraima (email: hugo8romao@gmail.com).

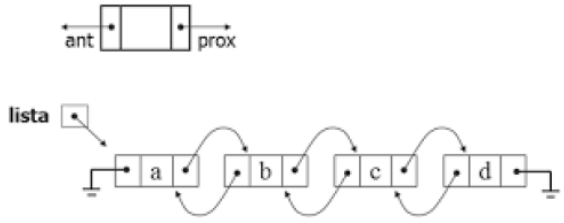
P. V. S. Ribeiro, Aluno na Universidade Federal de Roraima (UFRR), discente do Curso de Ciências da Computação, do Departamento de Ciências da Computação (DCC), Boa Vista - Roraima (email: silvapv7@gmail.com).

II. Fila de Prioridade

Uma estrutura auxiliar fundamental para a execução do algoritmo de Huffman é a fila de prioridade. Ao contrário da fila convencional, a fila de prioridade não segue a política de acesso First In First Out (FIFO). Nessa estrutura, cada elemento possui um atributo de prioridade e eles são ordenados com base neste atributo.

Normalmente, as filas de prioridade são implementadas utilizando vetores estáticos, e a remoção ocorre retirando-se o elemento de maior prioridade da fila. No entanto, para este projeto, o funcionamento da estrutura foi modificado para se adequar ao contexto específico.

Após a leitura dos dados do documento, um nó é criado na lista para cada caractere identificado. Se um caractere aparece várias vezes, sua prioridade é incrementada e o elemento é reordenado de acordo com a nova prioridade.



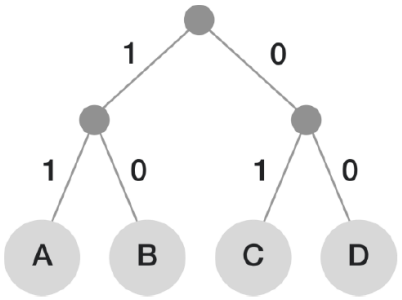
III. Árvore de Huffman

Consiste em uma Binary Search Tree (BST) cujos valores das folhas são exatamente os caracteres encontrados no documento lido. O caminho percorrido para chegar em determinado caractere determina a sua representação binária no arquivo compactado.

Criada a fila de prioridade, inicia-se a construção da árvore de Huffman. Enquanto existem elementos na fila os seguintes passos são realizados:

- Remove-se os dois nós de menor frequência da fila,
- Cria-se um novo nó cuja prioridade equivale a soma dos nós removidos,
- Os nós removidos são adicionados como filhos do novo nó,
- O nó resultante é inserido na primeira posição da fila e depois reordenado.

Após o procedimento, a árvore de Huffman vai estar completa.



IV. Tabela de Símbolos

É um array de strings de 256 posições criado para referenciar todos os símbolos da tabela ASCII a sua respectiva representação em bits. A tabela é construída percorrendo a árvore de Huffman recursivamente até que todos os nós folhas sejam encontrados.

A tabela abaixo é um exemplo de tabela de símbolos com algumas posições referenciadas.

Tabela de Símbolos	
A	0
B	101
C	100
D	111
E	1101
F	1100

V. Transformação dos dados e reconstrução

Tendo em vista a impossibilidade de gravar os bits diretamente no arquivo, a tradução da cadeia de caracteres em cadeias de bits equivalentes ganha uma camada extra de complexidade.

A solução encontrada para resolver tal problema consiste na construção de bytes de informação a partir da cadeia de bits gerada na tradução do documento pela tabela de símbolos.

Tomando como exemplo a tabela acima, podemos tentar construir a tradução da palavra *"cade"* que equivale a 10001111101 e possui 11 bits de comprimento. Dividindo os bits em cadeias de bytes temos 10001111 10100000 no formato compactado.

VI. Gravação dos dados

Os dados agrupados em bytes são salvos no arquivo junto com a extensão do arquivo original, o seu tamanho e a tabela de símbolos gerada para posterior recuperação. A extensão do arquivo compactado é ".hfz ".

VII. Compactação

O processo de compactação dos dados do documento consiste na seguinte sequência de passos:

- Leitura dos dados de entrada,
- Construção da fila de prioridade,
- Construção da árvore de Huffman,
- Construção da tabela de símbolos,
- Transformação do texto em cadeias de bits,
- Construção do fluxo de bytes comprimido,
- Gravação dos dados.

VIII. Descompactação

A descompactação do arquivo ".hfz" ocorre na realização dos seguintes passos:

- Leitura dos dados de entrada,
- Identificação da extensão,
- Reconstrução da árvore de Huffman,
- Reconhecimento do tamanho do arquivo original,
- Reconhecimento do conteúdo,
- Reconstrução do fluxo de bits,
- Transformação do conteúdo compactado para o conteúdo original,
- Gravação dos dados.

Durante a descompactação do arquivo, os procedimentos basicamente são o processo inverso do que ocorre na compactação, dito isso apenas alguns procedimentos merecem atenção especial, dentre eles:

A. A reconstrução da Árvore de Huffman:

Realizada através de um algoritmo recursivo que utiliza a tabela de símbolos como base.

B. A transformação do Conteúdo:

O algoritmo utiliza a Árvore de Huffman como base para realizar a tradução dos conjuntos de bits correspondentes a cada caractere. Durante esse processo, o programa lê bit a bit do arquivo compactado e percorre a árvore de acordo com o bit lido. Quando um nó folha é alcançado, o caractere associado a esse nó é reconhecido e armazenado no arquivo descompactado. Em seguida, o programa retorna ao topo da árvore e continua a leitura de bits até que todo o conteúdo do arquivo compactado tenha sido traduzido de volta para o conteúdo original.

IX. Análise do Algoritmo

A análise de complexidade do algoritmo de construção da tabela de símbolos utilizando a Trie de Huffman é de grande relevância entre as três estruturas auxiliares. A tabela de símbolos e a fila de prioridade, embora sejam utilizadas apenas para funções de tempo constante, não exercem um grande impacto na complexidade geral do algoritmo.

A seguir, apresentamos uma análise detalhada da complexidade do algoritmo recursivo de construção da tabela de símbolos utilizando a Trie de Huffman:

A. O Código

```
void Huff::buildCode(string* codes, Node* node, string code)
{
        if (node->isLeaf())
        {
                codes[node->getSymbol()] = code;
                return;
}
this->buildCode(codes, node->getLeft(), code + '0');
this->buildCode(codes, node->getRight(), code + '1');
}
```

B. Equação de Recorrência

$$\begin{cases} T(n) = 1, \ para \ n = 1 \\ T(n) = 2T\left(\frac{N}{2}\right) + 1, \ para \ n > 1 \end{cases} \quad (1)$$

- Pelo método da interação temos:

$$T(n) = 2T\left(\frac{N}{2}\right) + 1$$

$$T(n) = 2(2T\left(\frac{N}{4}\right) + 1) + 1$$

$$T(n) = 2(2(2T\left(\frac{N}{8}\right) + 1) + 1) + 1$$

$$T(k) = 2^k T\left(\frac{N}{2^k}\right) + \sum_{i=0}^{k-1} 2^i \quad (2)$$

- De volta ao Caso base

$$T(k) = 2^k T\left(\frac{2^k}{2^k}\right) + \sum_{i=0}^{k-1} 2^i$$

$$T(k) = 2^k + \sum_{i=0}^{k-1} 2^i$$

$$Para \ n = 2^k$$

$$T(n) = 2^{\log(n)} + \sum_{i=0}^{\log(n)-1} 2^i$$

$$T(n) = n + \frac{n}{2} - 1 \quad (3)$$

C. Complexidade

O melhor caso ocorre quando o elemento acessado é o da primeira posição da lista.

O pior caso é quando o elemento se encontra em uma parte degenerada da árvore, onde os elementos se distribuem como em uma lista ligada.

Complexidade	
Melhor Caso	O(1)
Caso Médio	O(log(n))

Pior Caso	O(n)

X. Testes

Para tal foram efetuados alguns testes para verificar tudo o que foi apresentado sobre o algoritmo de Huffman.

O teste foi feito usando máquinas com Hardwares e Arquiteturas completamente diferentes para que pudéssemos verificar o comportamento de algoritmo em diferentes situações.

Para avaliar o algoritmo proposto foram utilizadas duas entradas diferentes, a primeira descreve uma página HTML simples, a segunda é uma reprodução em português do famoso artigo de Alan Turing *"Computadores e Inteligência", de 1950*. A máquina utilizada na primeira rodada de testes é descrita a seguir.

A. Primeiro teste:

O primeiro teste foi feito na máquina que chamaremos de "máquina 01", a seguir as especificações da mesma:

máquina 01	
Modelo	MacBook Air
CPU	Apple M2
Base Frequency	3.5 GHz
RAM	8GB

A seguir, um teste de uma página HTML simples (basic.HTML) foi comprimida para demonstrar o funcionamento do algoritmo:.

- Comando para execução:
  g++ -std=c++11 main.cpp -o ./builds/main && ./builds/main ./tests/basic.html
- Descrição do teste:

Tempo de Construção da Tabela	0.000611 sec
Primeiro Texto original (120 bits)	BCAADDDCCACACAC
Primeiro Texto codificado (86 bits)	0100000101100111101111 1011101001101001101000 1100011001111010110011 1101011001111010110 0
Redução de	76.666666%

O segundo teste foi feito usando uma página HTML maior (turing.HTML), página com o artigo "Computadores e inteligência" de Alan Turing:

- Comando para execução:
  g++ -std=c++11 main.cpp -o ./builds/main && ./builds/main ./tests/turing.html
- Descrição do teste:

Tempo de Construção da Tabela	0.002041 sec
Segundo Texto (7512 bits)	Proponho a seguinte questão: "Podem as máquinas pensar?" A consideração disso deveria ser iniciada com definições do significado dos termos "máquinas" e "pensar".. (...)
Segundo Texto codificado (86 bits)	1010011000000010000001 0100010110101001011000 0111010100111110100100 0100111111110110101111 1 1001000111100111110001 1101110010000011001000 0010100111011111011110 0 (...)
Redução de	45.54287%

B. Segundo teste:

O Segundo teste foi feito na máquina que chamaremos de "máquina 02", a seguir as especificações da mesma:

máquina 02	
Modelo	Acer Nitro AN515-52
CPU	Intel Core i5-8300H
Base Frequency	2.30 GHz
RAM	8GB

A seguir, a repetição do primeiro teste da página HTML simples (basic.HTML) sendo usando na "máquina 02:

- Comando para execução:
  g++ -std=c++11 main.cpp -o ./builds/main && ./builds/main ./tests/basic.html
- Descrição do teste:

Tempo de Construção da Tabela	0.000818 sec
Primeiro Texto original (120 bits)	BCAADDDCCACACAC
Primeiro Texto	1101011101100100010101 1111101110100101010101

codificado (28 bits)	11000101001011
Redução de	76.666666%

Por fim a repetição do segundo teste usando uma página HTML maior (turing.HTML), página com o artigo “Computadores e inteligência" de Alan Turing na “máquina 02”:

- Comando para execução:
  g++ -std=c++11 main.cpp -o ./builds/main && ./builds/main ./tests/turing.html
- Descrição do teste:

Tempo de Construção da Tabela	0.000943 sec
Segundo Texto (7512 bits)	Proponho a seguinte questão: “Podem as máquinas pensar?” A consideração disso deveria ser iniciada com definições do significado dos termos “máquinas” e “pensar”.. (...)
Segundo Texto codificado (86 bits)	10101110100000100110110101011001100111000100001111111010111111111010011111111010011011110110011010010011011110110011100111111000011000110011111100101010111010001100111001111111000011000110 (...)
Redução de	45.54287%

Pensando na utilização deste algoritmo como forma de transmissão de informação entre dois atores, destaca-se a existência de um comportamento de compensação quando a entrada passada é muito pequena, isso ocorre pois além de enviar o conteúdo codificado, é necessário enviar a tabela de tradução pelo menos uma vez ao receptor da mensagem, para que seja possível decodificar as mensagens seguintes.

Podemos observar que o comportamento do algoritmo pode variar de acordo com o hardware da máquina em que está sendo executado. Para ilustrar essa variação, foram realizados testes tanto na "Máquina 01" quanto na "Máquina 02", e o algoritmo apresentou tempos de execução distintos.

XI. Conclusão

Em conclusão, o algoritmo de codificação/decodificação de Huffman destaca-se por sua simplicidade e eficácia, possibilitando altas taxas de compressão sem perda de dados. Ao longo das décadas, continua sendo um codec de extrema importância nas tecnologias da informação, sendo amplamente utilizado em aplicações profissionais e acadêmicas, além de ser incorporado em normas populares de multimídia, como JPEG, MPEG, MP3 e até mesmo HTML como foi mostrado neste estudo.

Apesar de suas vantagens, o algoritmo de Huffman também apresenta algumas desvantagens. Ele pode ser ineficiente ao codificar alfabetos de pequeno tamanho e exige a modelagem de probabilidades antes da codificação, o que pode afetar o processamento rápido do algoritmo, como dito anteriormente. Essas desvantagens podem levar à preferência pela codificação aritmética em certos contextos de compressão de entropia.

A elaboração deste artigo, juntamente com a implementação do codec de Huffman em uma aplicação C++, proporcionou uma análise detalhada de um dos algoritmos mais importantes do século XX, amplamente utilizado nas tecnologias atuais. A implementação prática permitiu uma compreensão completa dos aspectos teóricos, semânticos e estruturais, enriquecendo o processo de desenvolvimento.

Referências

Link do repositório para consulta: https://github.com/hugolima03/HugoLimaPedroVinicius_FinalProject_AA_RR_2023.git

[1] N. Ziviani, Projeto de Algoritmos com Implementações em Pascal e C, 3rd ed. Cengage Learning, 2011, iSBN: 978-85-221-1050-6.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, 3rd ed. The MIT Press, 2009, iSBN-13: 978-0-262-53305-8.

[3] D. Antoniades, I. Polakis, G. Kontaxis, E. Athanasopoulos, S. Ioannidis, E. Markatos, and T. Karagiannis, “we.b: The web of short urls,” in Int’l Conference on World Wide Web., 2011, pp. 715–724.

[4] Huffman, D. A. (1952). A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the I.R.E.*, 1098-1101.

[5] Blelloch, G. E. (2001). Introduction to Data Compression. In G. E. Blelloch, *Algorithms in the Real World.* Pittsburg, PA, United States of America: Carnegie Mellon University.

[6] Pu, I. M. (2006). *Fundamental Data Compression.* Oxford, United Kingdom: Elsevier.

APÊNDICES

A. Árvore gerada pela página "turing.html"