Relatório do Projeto de Compressor Huffman

Kauã do Vale Ferreira Caio de Medeiros Trindade

14 de outubro de 2025

1 Introdução

Este documento detalha a implementação de um compressor e descompressor de arquivos baseado no algoritmo de Huffman, desenvolvido como parte do projeto da disciplina de Estrutura de Dados 2. O objetivo principal foi criar uma ferramenta de linha de comando em C++ capaz de comprimir arquivos de texto, com uma otimização especial para códigos-fonte de linguagens como C++, PY, Java, JS, TS.

O projeto foi dividido em dois executáveis principais:

1. **Contador de Frequência**: Uma ferramenta para analisar um conjunto de arquivos e gerar uma tabela de frequência de "símbolos". Diferente de uma abordagem tradicional, os símbolos não são apenas caracteres individuais, mas também palavras-chave da linguagem (ex: `int`, `while`, `std::vector`), permitindo uma compressão mais semântica e eficiente para código.

2. **Compressor/Descompressor**: O programa principal que utiliza uma tabela de frequência (seja uma pré-calculada ou gerada em tempo real) para construir uma Árvore de Huffman, comprimir um arquivo de entrada e, inversamente, descomprimir um arquivo para restaurar seu conteúdo original.

2 Análise de Complexidade

A eficiência do compressor pode ser analisada em três etapas principais: a contagem de frequência, o processo de compressão e o de descompressão.

2.1 Contagem de Frequência

O processo de contagem de frequência, realizado pelo programa `contador_frequencia`, envolve a leitura de um ou mais arquivos de entrada e a contabilização dos símbolos.

- **Leitura e Tokenização**: O programa lê cada arquivo de entrada, de tamanho N, e o divide em tokens. A identificação de palavras-chave é feita com base em um conjunto pré-definido. Esta etapa percorre todo o conteúdo do arquivo uma vez.

- **Armazenamento**: As frequências são armazenadas em um `std::map`. A inserção ou atualização de um símbolo no mapa tem um custo médio de $O(\log K)$, onde K é o número de símbolos únicos.

A complexidade total desta etapa é dominada pela leitura e processamento do arquivo, resultando em $O(N \log K)$. Como o número de símbolos únicos (K) é geralmente muito menor que o tamanho do arquivo (N), a complexidade na prática se aproxima de $O(N)$.

2.2 Processo de Compressão

A compressão, executada pelo programa `compressor`, envolve os seguintes passos:

1. **Leitura da Tabela de Frequência**: Se um arquivo de tabela é fornecido, a leitura tem custo proporcional ao número de símbolos únicos, $O(K)$.

2. **Construção da Árvore de Huffman**: A árvore é construída a partir das frequências usando uma `std::priority_queue`. Inserir os K símbolos na fila de prioridade custa $O(K \log K)$. Em seguida, o algoritmo remove os dois nós de menor frequência e insere um novo nó pai, repetindo o processo $K - 1$ vezes. Cada operação (remoção/inserção) custa $O(\log K)$. Portanto, a complexidade total para construir a árvore é $O(K \log K)$.

3. **Geração dos Códigos**: O programa percorre a árvore para gerar o mapa de códigos binários para cada símbolo. A complexidade desta etapa é proporcional ao tamanho da árvore, $O(K)$.

4. **Codificação do Arquivo**: O arquivo de entrada, de tamanho N, é lido e tokenizado novamente (se necessário). Em seguida, cada token é substituído pelo seu código de Huffman correspondente. O custo é proporcional ao número de tokens (T) multiplicado pelo comprimento médio dos códigos (L_{avg}), resultando em $O(T \times L_{avg})$. No total, isso é proporcional ao tamanho do arquivo de saída.

A etapa dominante é a construção da árvore, tornando a complexidade geral da compressão $O(N + K \log K)$.

2.3 Processo de Descompressão

A descompressão é um processo mais direto:

1. **Desserialização da Árvore**: O programa lê o cabeçalho do arquivo comprimido para reconstruir a Árvore de Huffman. A complexidade é proporcional ao número de nós da árvore, $O(K)$.

2. **Decodificação**: O restante do arquivo é lido bit a bit. Para cada bit, o programa navega na árvore (da raiz para a esquerda ou direita) até encontrar um nó-folha, que corresponde a um símbolo. Este processo é repetido até o fim do arquivo. Se o arquivo comprimido tem M bits, a complexidade é $O(M)$.

A complexidade da descompressão é linear em relação ao tamanho do arquivo comprimido.

3 Análise da Taxa de Compressão

Para avaliar a eficácia do compressor, realizamos testes comparativos com diversos tipos de arquivos. Comparamos o desempenho do nosso compressor (`HuffmanCompressor`) com ferramentas de compressão padrão do mercado, como `zip` e `gzip`.

A taxa de compressão foi calculada usando a fórmula:

$$\text{Taxa de Compressão} = 1 - \left(\frac{\text{Tamanho Comprimido}}{\text{Tamanho Original}} \right)$$

3.1 Tabela de Resultados

Tabela 1: Comparação da Taxa de Compressão

Arquivo de Teste	Original	Ferramenta	Comprimido	Taxa (%)
C++ (complexo)	7.1 KB	**HuffmanCompressor**	**4.9 KB**	**31.0%**
`compressor.cpp`	7.1 KB	zip	5.3 KB	25.4%
Texto Simples	10.2 KB	HuffmanCompressor	6.1 KB	40.2%
`texto.txt`	10.2 KB	**zip (DEFLATE)**	**5.5 KB**	**46.1%**
C++ (simples)	312 B	HuffmanCompressor	283 B	9.3%
`simple_code.cpp`	312 B	**gzip**	**239 B**	**23.4%**
Java (médio)	980 B	HuffmanCompressor	631 B	35.6%
`medium_code.java`	980 B	**gzip**	**347 B**	**64.6%**
JavaScript (médio)	3.9 KB	HuffmanCompressor	2.7 KB	30.6%
`medium_code.js`	3.9 KB	**gzip**	**1.4 KB**	**63.2%**
TypeScript (simples)	431 B	HuffmanCompressor	350 B	18.8%
`simple_code.ts`	431 B	**gzip**	**292 B**	**32.3%**

3.2 Análise dos Resultados

Os resultados experimentais confirmam a hipótese central do projeto:

1. **Desempenho em Código-Fonte C++**: O `HuffmanCompressor` foi superior ao `zip` na compressão do arquivo de código-fonte C++ mais complexo. A taxa de compressão de 31.0% contra 25.4% do `zip` demonstra a vantagem da nossa abordagem. Ao tratar palavras-chave como `std::string`, `vector`, `if`, e `while` como símbolos únicos, nosso algoritmo atribui a eles códigos binários muito curtos.

2. **Desempenho em Texto Simples**: Para o arquivo de texto comum, o `zip` obteve um resultado superior. Isso ocorre porque algoritmos como o DEFLATE, usado pelo `zip`, combinam a codificação de Huffman com o algoritmo LZ77, que é extremamente eficaz em encontrar e substituir sequências de caracteres repetidas.

3. **Desempenho em Outros Códigos-Fonte**: Os testes com códigos em outras linguagens, usando uma tabela de frequência treinada apenas com C++, revelaram um insight interessante. Para arquivos pequenos ou linguagens com sintaxe muito diferente, o custo de serializar a árvore de Huffman e a falta de uma tabela de frequência otimizada fizeram com que ferramentas padrão como `gzip` tivessem um desempenho superior. Isso sugere que, para obter a máxima eficiência, a tabela de frequência deve ser treinada com um corpus representativo da linguagem-alvo.

3.3 Desafio de Implementação: O Bug do Padding de Bits

Durante os testes de validação, foi identificado um bug de caso de borda que afetava a integridade dos arquivos descomprimidos. Em alguns casos, notavelmente com um arquivo de teste em Java, o arquivo restaurado continha um caractere extra no final, fazendo com que a verificação de integridade (`diff`) falhasse.

Causa Raiz do Problema A causa do problema está na natureza da escrita de dados em bits. O algoritmo de Huffman gera códigos de comprimentos variáveis, mas os sistemas de arquivos operam em bytes (grupos de 8 bits). Quando o número total de bits do conteúdo comprimido não é um múltiplo de 8, a última escrita no arquivo preenche o byte final com bits de *padding* (preenchimento) que não fazem parte dos dados originais.

A versão inicial do nosso descompressor lia o fluxo de bits continuamente até que o stream terminasse. Consequentemente, ele tentava interpretar esses bits de padding como se fossem um código de Huffman válido, resultando na decodificação de um símbolo final incorreto.

Solução Implementada Para resolver este problema de forma robusta, modificamos o formato do nosso arquivo `.huff`. A solução consistiu em duas etapas:

1. **Na Compressão:** Antes de escrever a árvore serializada e os dados comprimidos, agora escrevemos um cabeçalho de 8 bytes contendo o **número total de símbolos (tokens)** que o arquivo original possuía.

2. **Na Descompressão:** O programa primeiro lê este número do cabeçalho. Em seguida, o loop de descompressão foi alterado de um laço infinito para um laço `for` que executa exatamente o número de vezes lido, garantindo que apenas a quantidade correta de símbolos seja decodificada.

Essa alteração tornou o processo de descompressão preciso, fazendo com que ele pare exatamente no último bit válido e ignore qualquer bit de padding, garantindo a restauração perfeita do arquivo original em todos os casos de teste.

4 Conclusão

O projeto foi concluído com sucesso, resultando em uma ferramenta de compressão robusta. A análise de desempenho mostrou que a especialização da tabela de frequência para incluir tokens de múltiplos caracteres é uma estratégia vitoriosa para a compressão de código-fonte. Embora não supere os compressores de uso geral em todos os cenários, o projeto prova que podemos sim, fazer algo semelhante que traga resultados bons.