Relatório - Projeto Huffman

Compressão de Arquivos com o Algoritmo de Huffman

Integrantes: Alice de Oliveira Duarte - 10419323
Pedro Roberto Fernandes Noronha - 10443434
Guilherme Silveira Giacomini - 10435311
Carlos Eduardo Diniz de Almeida - 10444407

---

1. Introdução

Este relatório documenta a implementação completa do algoritmo de Huffman para compressão e descompressão de arquivos sem perdas. O projeto demonstra a aplicação prática de estruturas de dados fundamentais - Fila de Prioridades (Min-Heap) e Árvore Binária - em um algoritmo clássico de compressão, validando sua eficácia através de análises detalhadas de performance e taxas de compressão.

2. Decisões de Projeto e Implementação das Estruturas

2.1 Arquitetura do Sistema

O sistema foi estruturado em três classes principais que encapsulam responsabilidades específicas:

`Huffman.java` - Classe principal contendo:

- Métodos `compress()` e `decompress()` para as operações principais
- Método `buildHuffTree()` para criar a árvore de Huffman usando o Min-Heap
- Método `buildCodes()` para gerar a tabela de códigos via percurso recursivo
- Método `measureElapsedTime()` para análise de performance usando `System.nanoTime()`

`MinHeap.java` - Implementação da Fila de Prioridades:

- Baseada em `ArrayList<No>` para flexibilidade de tamanho
- Métodos `siftUp()` e `siftDown()` para manter propriedade de heap
- Operações `insert()` e `removeMin()` com complexidade O(log n)
- Índices calculados como: pai = (i-1)/2, filhos = 2*i+1 e 2*i+2

`No.java` - Representação dos nós da árvore:

- Implementa `Comparable<No>` para ordenação por frequência

- Campos `caractere`, `frequencia`, `esquerda`, `direita`
- Método `ehFolha()` para identificação de nós terminais

2.2 Detalhes Técnicos de Implementação

Min-Heap com ArrayList: ArrayList é a escolha ideal porque oferece acesso direto aos elementos por índice com get() e set() em O(1), crescimento automático com add() quando necessário, e permite calcular posições pai/filhos facilmente usando fórmulas matemáticas simples. Isso torna as operações de heap mais eficientes que usar arrays fixos ou listas ligadas.

Array de Strings para códigos: `String[256]` oferece acesso direto O(1) para qualquer caractere ASCII, otimizando a fase de codificação.

Percurso recursivo para geração de códigos: Implementação natural que segue a estrutura da árvore, garantindo códigos únicos e livres de prefixo.

Manipulação de Bytes: O algoritmo trabalha com bytes (8 bits) usando `byte[]` para ler arquivos e `b & 0xFF` para converter bytes para índices de array. Esta abordagem é necessária porque Java trata bytes como valores com sinal (-128 a 127), mas precisamos de índices de array positivos (0 a 255). A operação `& 0xFF` remove o sinal e garante que todos os 256 valores ASCII possíveis sejam mapeados corretamente na tabela de frequências.

Escrita bit-by-bit: Durante a compressão, os códigos de Huffman são escritos bit por bit usando operações de shift (`<<`) e OR (`|`) para construir bytes completos. Quando não há bits suficientes para completar um byte, padding com zeros é adicionado automaticamente. Isso permite compressão eficiente onde códigos podem ter qualquer tamanho de 1 a vários bits.

3. Análises Requeridas

Parte 1: Análise de Performance (Tempo)

Utilizando `System.nanoTime()`, foram medidos os tempos de execução para diferentes tamanhos de arquivo:

Arquivo	Tamanho	Compressão (ms)	Descompressão (ms)	Razão C/D
BANANA	6 bytes	15.132	1.504	10.1:1
Código-fonte	917 bytes	21.789	2.000	10.9:1
Repetitivo	1,000 bytes	15.313	1.490	10.3:1
Random	1,024 bytes	31.548	2.520	12.5:1

Arquivo	Tamanho	Compressão (ms)	Descompressão (ms)	Razão C/D
arq_de_teste.txt	6,279 bytes	25.376	3.053	8.3:1

Análise do Crescimento Temporal

Compressão: O crescimento temporal é aproximadamente linear com o tamanho do arquivo, confirmando a complexidade teórica **O(n log k)** onde n é o tamanho do arquivo e k o número de caracteres distintos (máximo 256). Arquivos com maior diversidade de caracteres (como dados aleatórios) requerem mais tempo devido à construção de árvores mais complexas.

Descompressão: Apresenta crescimento linear **O(m)** onde m é o número de bits comprimidos, sendo consistentemente 8-12x mais rápida que a compressão, pois envolve apenas navegação na árvore sem construção de estruturas.

Os resultados confirmam que o algoritmo mantém performance previsível e escalável, adequada para uso prático.

Parte 2: Comparação de Taxas de Compressão (Espaço)

Aplicando a fórmula: Taxa = (1 - Tamanho_Comprimido/Tamanho_Original) × 100%

Tipo de Arquivo	Tamanho Original	Tamanho Comprimido	Taxa de Compressão	Redução
Texto comum (arq_de_teste.txt)	6,279 bytes	3,577 bytes	**43.04%**	2,702 bytes
Código-fonte (.java)	917 bytes	505 bytes	**44.97%**	412 bytes
Muito repetitivo (AAAA...)	1,000 bytes	125 bytes	**87.50%**	875 bytes
Caracteres aleatórios	1,024 bytes	1,003 bytes	**2.05%**	21 bytes

Análise das Variações de Taxa

Por que arquivos repetitivos comprimem bem (87.50%):

- Distribuição extremamente desigual de frequências
- Caractere dominante ('A') recebe código de 1 bit versus 8 bits originais
- Poucos caracteres distintos resultam em árvore simples e códigos curtos
- Representa o cenário ideal para o algoritmo de Huffman

Por que texto comum tem boa compressão (43.04%):

- Padrões naturais: espaços, vogais e consoantes frequentes
- Caracteres comuns (espaço, 'a', 'e') recebem códigos curtos (2-3 bits)
- Caracteres raros recebem códigos mais longos, otimizando o resultado global

Por que código-fonte comprime bem (44.97%):

- Palavras-chave repetidas (public, class, void, etc.)
- Operadores e símbolos comuns (';', '(', ')', '=')
- Estruturas sintáticas previsíveis aumentam repetição
- Similar ao texto natural mas com vocabulário mais restrito

Por que dados aleatórios têm compressão ruim/negativa (2.05%):

- Distribuição aproximadamente uniforme de todos os 256 caracteres possíveis
- Códigos de Huffman têm comprimento similar (~8 bits) ao ASCII original
- **Overhead do cabeçalho** (256 × 4 = 1,024 bytes) pode causar expansão
- Para arquivos pequenos aleatórios, o resultado pode ser maior que o original

Esta análise demonstra que Huffman é eficiente quando existe desbalanceamento na distribuição de frequências e ineficiente para dados uniformemente distribuídos, validando a teoria de informação.

4. Validação e Verificação

4.1 Teste com Exemplo da Documentação

O algoritmo foi validado com o exemplo "BANANA" especificado:

Resultado Esperado vs Obtido:

- Frequências: B=1, A=3, N=2 (correto)
- Códigos: A='0', B='10', N='11' (correto)
- Compressão: 48→9 bits (81.25%) (correto)
- Resultado idêntico à especificação

4.2 Verificação de Integridade

Todos os arquivos descomprimidos foram verificados como binariamente idênticos aos originais usando `fc /B`, confirmando:

- Compressão sem perdas (lossless)
- Implementação correta dos algoritmos
- Robustez das estruturas de dados

4.3 Conformidade com Especificação

- Formato de saída no console conforme especificado (5 ETAPAs)
- Execução via linha de comando `java -jar huffman.jar -c/-d`
- Arquivos originais preservados, novos arquivos gerados
- Min-Heap e Árvore Binária implementados corretamente

5. Conclusões

5.1 Eficácia do Algoritmo

A implementação demonstrou que o algoritmo de Huffman mantém sua relevância prática:

- Excelente para dados com padrões (87.50% compressão)
- Bom para texto e código natural (43-45% compressão)
- Inadequado para dados uniformes (2.05% compressão)

5.2 Qualidade da Implementação

- Performance conforme teoria: Complexidades O(n log k) e O(m) respeitadas
- Robustez comprovada: Funciona em todos os cenários testados