Universidade Federal do Rio Grande do Norte
Departamento de Informática e Matemática Aplicada
Centro de Ciências Exatas e da Terra

Relatório Técnico: Análise de Complexidade do Algorítmo de Huffman

Relatório Técnico apresentado como avaliação da matéria de Estrutura de Dados Básicas II ministrada no Departamento de Informática e Matemática Aplicada do Centro de Ciências Exatas e da Terra da Universidade Federal do Rio Grande do Norte - campus Central, conduzida pelo professor Dr. André Maurício Cunha Campos.

Natal, RN
2025

THIAGO ELIAS SOUZA MATTAR
THIAGO LOPES MARTINS

Relatório Técnico: Análise de Complexidade do Algorítmo de Huffman

Relatório Técnico apresentado como avaliação da matéria de Estrutura de Dados Básicas II ministrada no Departamento de Informática e Matemática Aplicada do Centro de Ciências Exatas e da Terra da Universidade Federal do Rio Grande do Norte - campus Central, conduzida pelo professor Dr. André Maurício Cunha Campos.

Natal, RN
2025

SUMÁRIO

1. INTRODUÇÃO

Este relatório tem o objetivo de apresentar a análise de complexidade temporal das funções que constroem o algoritmo de Huffman (ou árvore de Huffman). O algoritmo funciona através da utilização de uma árvore binária com o objetivo de comprimir dados, nesse caso de um arquivo de código-fonte, de forma que seu funcionamento é através da alocação de poucos bits para caracteres que tem muita repetição e mais bits para aqueles que se repetem menos. Durante o relatório, serão analisadas as complexidades temporais das funções que fazem parte do algoritmo de Huffman, utilizando a notação Big O (pior caso). Logo em seguida, amostras de testes coletando os tamanhos de arquivos antes e depois da compressão, comprovando, dessa forma, sua eficácia.

2. ANÁLISE DE COMPLEXIDADE DO ALGORITMO DE HUFFMAN

```
std::vector<Token> parse_tokens(const std::string& filename)
```

Custo: O(n), apenas o laço principal de while tem custo n. Todas as operações dentro do laço também terão custo n.

```
static bool is_symbol(char c)
```

Custo: constante, O(1).

```
 void write_bit(int b)
```

Custo: constante, O(1).

```
void write_bits(const std::string &bits)
```

Custo: O(n).

```
void flush()
```

Custo: constante, O(1).

```
int read_bit()
```

Custo: constante, O(1).

```
HuffmanNode* buildHuffmanTree(const std::unordered_map<std::string,
uint64_t>& freqMap)
```

Custo: O(s logs), com a inserção na árvore com tamanho (log s) de s símbolos, dessa forma, o custo é s logs.

```
void freeTree(HuffmanNode* node) {
```

Custo: constante, O(1), simples destrutor.

```
void FreqCounter::add(const std::string& token)
void FreqCounter::add(const Token& token)
```

Custo: ambos são O(1), ou seja, constantes.

```
void FreqCounter::print() const
```

```
void FreqCounter::saveCSV(const std::string& filename) const
```
Custo: depende da quantidade de tokens.

```
void generateCodes(HuffmanNode* root, const string& prefix,
unordered_map<string, string>& codes) {
```
Custo: O(n logn), pois realiza a concatenação de strings n vezes.

```
void compressFile(const string& inputFile, const string& outputFile,
                  const unordered_map<string, string>& codes)
```
Custo: O(n²), pois há o uso da função write_bits de custo n dentro do laço de custo n.

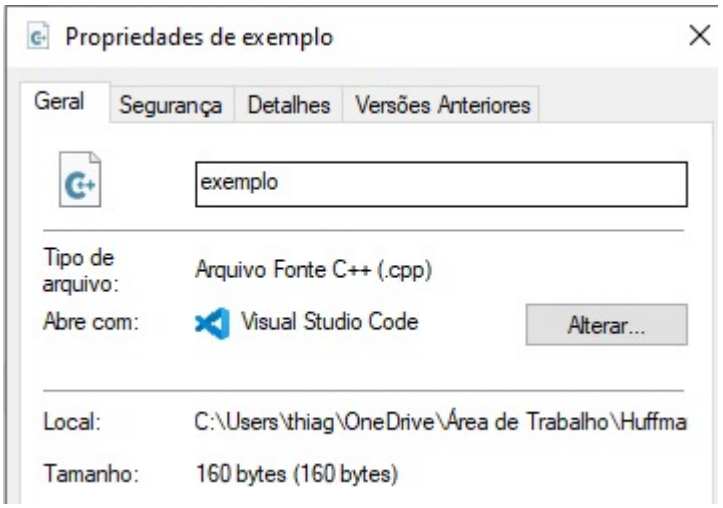
```
void decompressFile(const string& inputFile, const string& outputFile,
HuffmanNode* root)
```
Custo: O(n), apenas o valor de ler os nós e reescrever os caracteres.

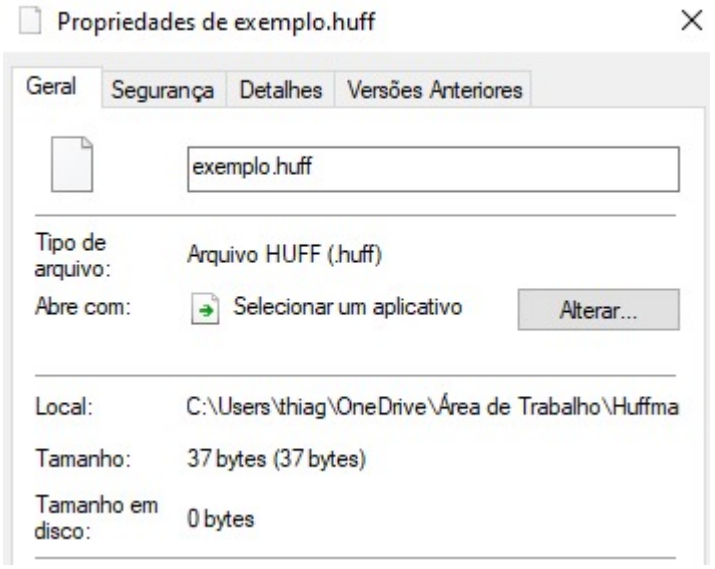
```
int main(int argc, char* argv[]) {
```
módulo c (compressão): O(n + s logs), a função de compressão tem valor de custo n e a construção da árvore tem valor de custo s logs.

módulo d (descompressão): mesma coisa que acima, sendo que a função de descompressão que tem valor n.

3. CONCLUSÃO

Concluímos que o algoritmo funciona, os arquivos realmente foram comprimidos, até com testes isso funciona. E vemos a funcionalidade do algoritmo ao fazer a tabela de quantidade de repetição dos caracteres. Abaixo há imagens do mesmo arquivo antes e depois da compressão.





Dessa forma, está provado que o algoritmo funciona, sendo o primeiro arquivo antes da compressão, com 160 bytes de tamanho, e o segundo arquivo é o pós compressão, com 37 bytes de tamanho.