Estrutura de Dados Básica II

Implementação de Compressão de Arquivos utilizando o Algoritmo de Huffman em C++

André Lucas Gonçalves Gomes
Yasmin Giordano Santos

Natal/RN
15 de outubro de 2025

Sumário

1 Resumo

Este projeto implementa um sistema de compressão e descompressão de arquivos baseado no algoritmo de Huffman, desenvolvido em C++ com recursos da STL e programação imperativa estruturada. A árvore de Huffman foi construída a partir de uma fila de prioridade implementada manualmente com MinHeap, garantindo que os símbolos e palavras mais frequentes recebam códigos binários mais curtos. O sistema inclui um contador de frequência de símbolos e palavras-chave e um compressor/descompressor que utiliza essa tabela para codificar e decodificar textos.

2 Introdução

A compressão de dados é uma técnica essencial na ciência da computação, amplamente utilizada para otimizar o armazenamento e a transmissão de informações. Seu objetivo é representar os dados de forma mais compacta, reduzindo o espaço ocupado em disco ou na memória sem comprometer a integridade das informações originais. Entre os diversos métodos de compressão existentes, o algoritmo de Huffman, proposto por David A. Huffman em 1952 [20], destaca-se como uma das abordagens mais eficientes para compressão sem perda, atribuindo códigos binários mais curtos a símbolos mais frequentes e códigos mais longos aos menos frequentes.

Este projeto tem como propósito implementar, em `C++`, um sistema completo de compressão e descompressão de dados baseado nesse algoritmo. Para melhor melhor estudo e compreensão, este trabalho foi dividido em dois módulos principais:

- **Contador de frequência:** responsável por analisar arquivos e gerar uma tabela de frequências de símbolos e palavras-chave, que serve de base para a construção da árvore de Huffman;
- **Compressor/Descompressor:** utiliza a tabela gerada para codificar e decodificar textos, transformando-os em representações compactas e restaurando o conteúdo original quando necessário.

Desse modo, é possível isolar as etapas de análise estatística e compressão, facilitando o estudo e visualização dos diferentes métodos implementados, além de contribuir para a possibilidade de reutilização da base de dados de frequências em diferentes contextos.

3 Objetivos

O objetivo principal deste projeto é aplicar os conhecimentos teóricos e práticos adquiridos na disciplina de **Estrutura de Dados Básica II (EDBII)**, com ênfase na implementação e integração de diferentes estruturas de dados, como árvores bináriaas (em especial as árvores de Huffman), filas de prioridade e min-heaps.

A proposta consiste em empregar essas estruturas na resolução de um problema concreto: a compressão de dados sem perda. Dessa forma, o projeto busca demonstrar, de maneira prática, a relevância do estudo de estruturas de dados e da manipulação de bits para o desenvolvimento de soluções computacionais eficientes, além de aprofundar a compreensão sobre o funcionamento e a aplicação do algoritmo de Huffman.

Mais especificamente, o projeto tem como objetivos:

- Implementar um **contador de frequência** capaz de identificar e registrar a ocorrência de símbolos e palavras-chave em arquivos;
- Construir e percorrer uma **árvore de Huffman** a partir das frequências obtidas, atribuindo códigos binários únicos a cada símbolo;
- Desenvolver uma **fila de prioridade**, estruturada manualmente como um **min-heap**, para gerenciar a combinação dos nós durante a construção da árvore;

- Criar um **compressor** e um **descompressor** que utilizem os códigos gerados pela árvore de Huffman para reduzir o tamanho de arquivos e restaurar o conteúdo original;
- Consolidar o entendimento de conceitos relacionados a **estruturas de dados**, **manipulação de bits** e **persistência em arquivos binários**, aplicando-os de forma integrada a um problema prático.

Em conjunto, esses objetivos visam fortalecer a compreensão sobre o funcionamento interno de diferentes estruturas de dados e dos algoritmos de compressão, além de evidenciar a importância da representação eficiente de arquivos em sistemas computacionais.

4 Fundamentação Teórica

4.1 Compressão de Dados

A compressão de dados é uma técnica fundamental na ciência da computação, utilizada para reduzir o espaço necessário para armazenar ou transmitir informações, sem perder a integridade dos dados originais. Ela pode ser classificada em dois tipos principais: compressão com perda, em que parte da informação é descartada para reduzir o tamanho, e compressão sem perda, que permite reconstruir os dados exatamente como eram antes da compressão [21].

A compressão sem perda é particularmente relevante em contextos onde cada símbolo é significativo, como em arquivos de código-fonte ou documentos de texto, tornando o algoritmo de Huffman uma escolha adequada para este projeto.

4.2 Algoritmo de Huffman

O algoritmo de Huffman, proposto por David Huffman em 1952 [20], é uma técnica clássica de compressão sem perda baseada em codificação de comprimento variável. O princípio central é atribuir códigos binários mais curtos aos símbolos mais frequentes e códigos mais longos aos símbolos menos frequentes.

O algoritmo envolve os seguintes passos:

1. **Contagem de frequências:** Determinar a frequência de cada símbolo no conjunto de dados.
2. **Construção da árvore de Huffman:** Criar uma árvore binária em que cada nó folha representa um símbolo, e cada caminho da raiz até uma folha representa o código binário desse símbolo. A árvore é construída combinando iterativamente os dois nós de menor frequência até formar um único nó raiz.
3. **Geração dos códigos:** Atribuir '0' e '1' a cada ramo da árvore, garantindo que nenhum código seja prefixo de outro, o que permite a decodificação sem ambiguidade.

O resultado é uma tabela de códigos binários otimizada, capaz de reduzir significativamente o tamanho do arquivo original, principalmente quando há alta repetição de certos símbolos.

4.3 Árvores Binárias e Filas de Prioridade

Árvores binárias são estruturas de dados fundamentais, compostas por nós em que cada nó pode ter no máximo dois filhos. Elas permitem a implementação de percursos eficientes, como pré-ordem, pós-ordem e em-ordem, que apresentam muita utilidade para a geração e leitura da árvore de Huffman [20].

Já na parte de construção da árvore de Huffman, é necessário escolher repetidamente os dois nós de menor frequência. Essa operação é otimizada por uma fila de prioridade, que mantém os elementos ordenados de acordo com sua frequência. Para esse contexto, foi escolhida a implementação de filas de

prioridade a partir de min-heap [22], estrutura de árvore binária completa que garante que o elemento de menor valor esteja sempre na raiz. Desse modo, a combinação dessas estruturas permite a construção eficiente da árvore e a geração correta dos códigos binários.

4.4 Manipulação de Bits e Arquivos Binários

Para armazenar os dados comprimidos de maneira eficiente, é necessário trabalhar diretamente com bits, em vez de apenas caracteres ou bytes. Cada símbolo codificado é representado por uma sequência de bits de comprimento variável, e várias sequências podem ser concatenadas em bytes para gravação em arquivo binário.

Essa abordagem requer atenção à manipulação de bits, deslocamento (shift) e operações lógicas para empacotar os códigos em bytes e, posteriormente, reconstruir os símbolos originais durante a descompressão. O uso de arquivos binários com `fstream` em C++ permite gravar e ler esses dados de forma eficiente, preservando a integridade da informação.

5 Desenvolvimento e Implementação

Nesta seção, são detalhados os principais aspectos de implementação do projeto, incluindo a descrição funcional dos módulos desenvolvidos, a estrutura e o comportamento das funções principais, bem como a análise de complexidade de cada uma delas.

A análise de complexidade será conduzida com base nos princípios da análise assintótica de algoritmos, considerando o comportamento das funções em relação ao tamanho da entrada.

5.1 Contador de Frequência

A função `main` do `Contador de Frequência` atua como ponto de entrada do programa, coordenando a execução das demais funções responsáveis por analisar o arquivo de entrada e gerar a tabela de frequências.

Parâmetros

`argc` : Número de argumentos da linha de comando.

`argv` : Vetor contendo os argumentos passados ao programa.

Retorno

Código de retorno (0 para sucesso, 1 para erro)

Listing 1: main.cpp

```cpp
int main (int argc, char* argv[]){
    if(argc != 2){
        cerr << "Error: incorrect number of arguments!" << endl;
        return 1;
    }
    string path = argv[1];

    getFullCharMap();

    // if path is a directory, process every .cpp file inside
    fs::path p(path);
    if (fs::exists(p) && fs::is_directory(p)) {
        for (const auto& entry : fs::directory_iterator(p)) {
            if (!entry.is_regular_file()) continue;
            if (entry.path().extension() == ".cpp" || entry.path().extension()
                == ".hpp") {
                processFile(entry.path().string());
            }
        }
```

```
    } else {
        // single file
        processFile(path);
    }

    sortedTokens = unifyAndSort(stringFrequency, charFrequency);

    printToFile("output");

    return 0;
}
```

Ao visualizar a função `main`, torna-se evidente que, para compreender o desempenho total do contador de frequência, é necessário analisar a complexidade de cada uma das funções auxiliares individualmente, considerando agora o processamento de múltiplos arquivos em um diretório. Essa análise permitirá aplicar o **Teorema da Adição**, que estabelece que, quando um algoritmo realiza uma série de operações ou funções de forma sequencial, a complexidade total é a soma das complexidades de cada etapa.

Como todos os arquivos com extensões `.cpp` e `.hpp` são processados sequencialmente. A complexidade total do programa depende do número de arquivos no diretório.

Podemos formalizar a complexidade considerando os dois cenários:

$$T_{\texttt{main}} = T_{\text{verificação argumentos}} + T_{\texttt{getFullCharMap}} + \max \begin{cases} \sum_{i=1}^{F} T_{\texttt{processFile}_i} & \text{(diretório)} \\ T_{\texttt{processFile}} & \text{(arquivo único)} \end{cases} + T_{\texttt{unifyAndSort}} + T_{\texttt{printToFile}}$$

Onde:

- F = número de arquivos `.cpp` e `.hpp` no diretório
- $T_{\texttt{processFile}_i}$ = complexidade para processar o i-ésimo arquivo
- A complexidade total agora é proporcional ao número de arquivos processados

Portanto, nas subseções seguintes, cada função será detalhada quanto ao seu funcionamento interno e à complexidade temporal, considerando o tamanho das entradas processadas e o cenário de múltiplos arquivos. Com base nessa análise, será possível determinar a complexidade assintótica total do módulo, justificando formalmente o comportamento do programa em relação ao número de arquivos de entrada, seus tamanhos individuais e o número de símbolos distintos processados.

5.1.1 Função `getFullCharMap()` - Criação de tabela de símbolos

A função `getFullCharMap()` tem como objetivo inicializar um mapa (`charFrequency`) que contém todos os caracteres ASCII "imprimíveis"(código 32 a 126), além de caracteres de controle mais comuns (como o `\n`), adicionados ao mapa de forma "imprimível".

Listing 2: getFullCharMap() - utils.cpp

```
void getFullCharMap() {
    // Caracteres de controle importantes
    const vector<string> controlChars = {"\\a", "\\b", "\\t", "\\n", "\\v", "\\f
        ", "\\r"};
    for (const auto& key : controlChars) {
        charFrequency[key] = 0;
    }

    // ASCII de 32 a 126
    for (int i = 32; i < 127; ++i) {
```

```
        char c = static_cast<char>(i);
        string key = string(1, c);
        charFrequency[key] = 0; // frequência inicial zero
    }
}
```

Podemos formalizar a análise utilizando o **Teorema da Adição** e o **Teorema da Multiplicação**:

$$T_{\texttt{getFullCharMap}} = T_{\text{inicialização de vector}} + \sum_{i=0}^{7} T_{\text{inserção no unordered\_map}} + \sum_{j=32}^{126} (T_{\text{inserção no unordered\_map}} + T_{cast} + T_{string})$$

Desse modo, sabendo que:

- Inicializar um vetor com elementos constantes em C++ é feito em tempo proporcional ao número de elementos e ao tamanho de cada elemento. Como a complexidade de copiar cada string para o vetor é **O(m)** (m := o tamanho máximo de cada string) e o vetor está sendo inicializado com 7 elementos de tamanho constante 2,

$$T_{\text{inicialização de vector}} = \sum_{i=0}^{7} O(2) = 7 \cdot O(2) = O(1)$$

  [25][24].

- $T_{\text{inserção no unordered\_map}} = \text{O}(1)$ [23].
- $T_{cast}(1) = O(1)$ já que é uma conversão de tipo primitivo (de `int` para `char`), ou seja, é uma operação elementar em hardware, que envolve apenas atribuição e truncamento de bits [27].
- $T_{string}(1) = O(1)$ já que é uma operação que aloca memória e copia o caractere n vezes ($T_{string}(n) = O(n)$) [26].

Podemos afirmar então que a complexidade total da função é:

$$T_{\texttt{getFullCharMap}} = O(1) + \sum_{i=0}^{7} O(1) + \sum_{j=32}^{126} \big(O(1) + O(1) + O(1)\big) = O(1)$$

5.1.2 Função `processFile()` - Processamento de arquivo e contagem de frequências

A função `processFile()` processa um arquivo de texto, contando frequências de caracteres e identificadores, com tratamento especial para comentários e strings. Além disso, nessa função ocorre a manipulação de dois mapas: um onde as chaves são caracteres (`charFrequency`, previamente populado em `getFullCharMap()`) e outro onde são strings (`stringFrequency`). No caso do primei

Parâmetros

`filename`: Nome do arquivo que deve ser lido para contagem de frequência.

Listing 3: processFile() - frequencyCounter.cpp

```
void processFile(const string& filename){
    ifstream file(filename);
    if (!file.is_open()) {
        cerr << "Error: unable to open '" << filename << "'." << endl;
        return;
    }

    string fullText((istreambuf_iterator<char>(file)), istreambuf_iterator<char
        >());
```

```
9       file.close();
10      size_t totalChars = fullText.length();
11
12      for (size_t i = 0; i < totalChars; i++) {
13          char currentChar = fullText[i];
14          if (isalnum(currentChar) || currentChar == '_'){
15              string word = extractWord(fullText, i);
16              stringFrequency[word]++;
17              i += word.length() - 1;
18              continue;
19          }
20          size_t j = ignoreCommentOrString(fullText, i);
21          if (j != string::npos) {
22              i = j - 1;
23              continue;
24          }
25          charFrequency[getPrintableChar(fullText[i])]++;
26      }
27
28      for (auto it = stringFrequency.begin(); it != stringFrequency.end(); ){
29          if (it->second < MIN_FREQUENCY){
30              for (char c : it->first)
31                  charFrequency[string(1, c)] += it->second;
32              it = stringFrequency.erase(it);
33          }
34          else ++it;
35      }
36  }
```

De forma análoga à função anterior, podemos formalizar a análise utilizando o **Teorema da Adição** e o **Teorema da Multiplicação**:

$$T_{\texttt{processFile}} = T_{\text{abertura de arquivo}} + T_{\text{leitura}} + \sum_{i=0}^{n} \left(T_{\text{operator[]}} + \max \begin{cases} T_{\text{string}} \\ T_{\text{comentário}} \end{cases} + T_{\text{getPrintableChar}} + T_{\text{acesso no unordered\_map}} \right) + \sum_{k=0}^{m} T_{\text{filtro}}$$

Onde:

- $T_{\text{abertura de arquivo}} = O(1)$ considerando que é uma operação de abertura de arquivo é operação de Entrada e saída constante [28].
- $T_{\text{leitura}} = O(n)$ onde n é o número de caracteres no arquivo (leitura com `istreambuf_iterator`) [29].
- $T_{\text{operator[]}} = O(1)$ [30].
- $T_{\text{string}} = T_{extractWord} + T_{\text{acesso no unordered\_map}} = T_{extractWord} + O(1) = T_{extractWord}$ [23].
- $T_{\text{comentário}} = T_{ignoreCommentOrString}$.
- $T_{\text{acesso no unordered\_map}} = \text{O(1)}$ [23].
- $T_{\text{filtro}} = O(m \cdot l)$ onde m é o número de entradas no mapa e l o comprimento das palavras.

Desse modo, para uma análise eficiente, precisamos primeiro chegar na complexidade temporal das funções auxiliares utilizadas.

A função `getPrintableChar` é responsável por transformar códigos em ASCII de caracteres em strings "imprimível".

Parâmetros

`char c`: Caractere em código ASCII.

Retorno

`string`: String "imprimível"equivalente ao caractere c.

Listing 4: getPrintableChar(char c) - utils.cpp

```
string getPrintableChar(char c) {
    switch (c) {
        case '\a': return "\\a";
        case '\b': return "\\b";
        case '\t': return "\\t";
        case '\n': return "\\n";
        case '\v': return "\\v";
        case '\f': return "\\f";
        case '\r': return "\\r";
        default:
            if (static_cast<unsigned char>(c) < 32 || c == 127) {
                return "\\x" + to_string(static_cast<int>(c));
            } else {
                return string(1, c);
            }
    }
}
```

$$T_{\texttt{getPrintableChar}} = T_{\text{switch}} + \max \begin{cases} T_{\text{retorno constante}} \\ T_{\text{conversão hexadecimal}} \end{cases}$$

- $T_{\text{switch}} = O(1)$ [31].
- $T_{\text{retorno constante}} = T_{\text{construir string}} = O(1)$ já que é uma criação de string com caractere único [26].
- $T_{\text{conversão hexadecimal}} = \max \begin{cases} T_{\text{construir string}} + T_{\text{static\_cast}} + T_{\text{to\_string}} + T_{\text{concatenar string}} \\ T_{\text{to\_string}} \end{cases} = T_{\text{construir string}} + T_{\text{static\_cast}} + T_{\text{to\_string}} + T_{\text{concatenar string}} = O(1) + O(1) + O(l) + O(2 + l)$ onde l é a quantidade de caracteres do inteiro em código ASCII, ou seja, 1 ou 2, portanto $T_{\text{conversão hexadecimal}} = O(1) + O(1) + O(1) + O(2) + O(4) = O(1)$ [33][26][32][27].

Portanto, temos que $T_{\text{getPrintableChar}} = O(1)$

A função `extractWord` é responsável por extrair uma palavra que começa a partir do caractere de entrada, o que permite que palavras-chave também sejam consideradas na contagem de frequência.

Parâmetros

`const string& text`: Texto em análise que contém o caractere de interesse.

`size_t pos`: Posição no texto recebido do caractere de interesse.

Retorno

`string`: Palavra que inicia a partir do caractere na posição `pos` do texto `text`.

Listing 5: extractWord() - utils.cpp

```
string extractWord(const string& text, size_t pos){
    string word = "";
```

```
    while (pos < text.length() &&
    (isalnum(static_cast<unsigned char>(text[pos])) ||
    text[pos] == '_')) {
        word += text[pos];
        pos++;
    }
    return word;
}
```

$$T_{\texttt{extractWord}} = \sum_{i=pos}^{text.length()-1} \left(T_{\text{verificação}} + T_{\text{concatenação}}\right)$$

- $T_{\text{verificação}} = O(1)$ já que `length()`, `isalnum()`, `static_cast`, `operator[]` e comparação têm complexidade temporal constante. [35]
- $T_{\text{concatenação}} = O(l)$ onde l é o tamanho total da palavra que está sendo lida. [33]

Portanto, temos que $T_{\text{extractWord}} = O(l)$.

A função `ignoreCommentOrString` é responsável por calcular o quanto será necessário "pular"do texto para sair de um caractere presente dentro de um comentário ou de uma string. Como tais ocorrências podem "sujar"a tabela de frequência, principalmente em um compressor que tem como alvo principal arquivos de código fonte, essa função é responsável por evitar tal sujeira.

Parâmetros

`const string& text`: Texto em análise que contém o caractere de interesse.

`size_t pos`: Posição do caractere que será analisado em busca do início de um comentário ou string.

Retorno

`size_t`: Posição no texto original em que o comentário ou string termina ou `npos` caso o caractere na posição `pos` não indique início de nenhuma sequência de texto nesse modelo.

Listing 6: ignoreCommentOrString() - utils.cpp

```
size_t ignoreCommentOrString(const string& text, size_t pos){
    size_t textSize = text.size();
    if (pos + 1 < textSize && text.substr(pos, 2) == "//") {
        stringFrequency["//"]++;
        size_t end = text.find('\n', pos + 2);
        if (end == string::npos) return textSize;
        charFrequency["\\n"]++;
        return end + 1;
    }
    if (pos + 1 < textSize && text.substr(pos, 2) == "/*") {
        stringFrequency["/*"]++;
        size_t end = text.find("*/", pos + 2);
        if (end == string::npos) return textSize;
        stringFrequency["*/"]++;
        return end + 2;
    }
    if (text[pos] == '"') {
        charFrequency["\""]++;
        size_t end = text.find('"', pos + 1);
        if (end == string::npos) return textSize;
        charFrequency["\""]++;
        return end + 1;
```

```
23      }
24      return string::npos;
25  }
```

$$T_{\texttt{ignoreCommentOrString}} = \max \begin{cases} T_{\text{comparação comentário linha}} + T_{\text{find}} \\ T_{\text{comparação comentário linha}} + T_{\text{comparação comentário bloco}} + T_{\text{find}} \\ T_{\text{comparação comentário linha}} + T_{\text{comparação comentário bloco}} + T_{\text{operator[]}} + T_{\text{find}} \\ T_{\text{comparação comentário linha}} + T_{\text{comparação comentário bloco}} + T_{\text{operator[]}} + T_{\text{retorno npos}} \end{cases}$$

Como já discutido anteriormente, todas essas comparações acabam tendo complexidade temporal constante ($O(1)$), assim como o $T_{\text{retorno npos}}$. Logo, o que define de fato $T_{\texttt{ignoreCommentOrString}}$ é T_{find}, ou seja, $O(n)$ [36].

Portanto, considerando:

- n: número total de caracteres no arquivo
- $l_{\max}$: comprimento máximo de uma palavra
- $c_{\max}$: comprimento máximo de comentário/string
- m: número de entradas no mapa `stringFrequency`

$$\begin{aligned} T_{\texttt{processFile}} &= O(1) + O(n) + \sum_{i=0}^{n} \left[O(1) + \max \begin{cases} T_{\text{extractWord}} \\ T_{\text{ignoreCommentOrString}} \end{cases} + T_{\text{getPrintableChar}} + O(1) \right] + \sum_{k=0}^{m} O(m \cdot l_{\max}) \\ &= O(1) + O(n) + \sum_{i=0}^{n} \left[O(1) + O(l_{\max}) + O(c_{\max}) \right] + O(m \cdot l_{\max}) \\ &= O(n) + \sum_{i=0}^{n} \left[O(l_{\max}) + O(c_{\max}) \right] + O(m \cdot l_{\max}) \end{aligned}$$

No pior caso onde $l_{\max} \approx n$ e $c_{\max} \approx n$, temos:

$$T_{\texttt{processFile}} = O(n) + O(n^2) + O(m \cdot n) = O(n^2)$$

Porém, na prática:

- Palavras e comentários têm tamanho limitado ($l_{\max}, c_{\max} \ll n$)
- O número de palavras únicas m é tipicamente $O(n/l_{\text{avg}})$

Portanto, a complexidade esperada é:

$$T_{\texttt{processFile}} = O(n) \quad \text{(caso típico)}$$

5.1.3 Função `unifyAndSort()` - Unificação e ordenação de tokens

A função `unifyAndSort()` consolida e ordena os tokens de caracteres e strings por frequência.

Parâmetros

`const unordered_map<string, int> charFrequency`: Mapa de frequência de caracteres do texto.
`const unordered_map<string, int> stringFrequency`: Mapa de frequência de strings do texto.

Retorno

`vector<pair<string, int>>`: Vetor de pares `(chave x frequência)` ordenados.

Listing 7: unifyAndSort() - main.cpp

```
vector<pair<string, int>> unifyAndSort(const unordered_map<string, int>
    charFrequency, const unordered_map<string, int> stringFrequency){
    vector<pair<string, int>> allTokens;

    for (const auto& pair : charFrequency)
        allTokens.push_back(pair);

    for (const auto& pair : stringFrequency)
        allTokens.push_back(pair);

    sort(allTokens.begin(), allTokens.end(), sortByFrequency);

    return allTokens;
};
```

Podemos novamente formalizar a análise utilizando o **Teorema da Adição**:

$$T_{\texttt{unifyAndSort}} = T_{\text{inicialização}} + \sum_{i=0}^{n} T_{\text{push\_back}} + \sum_{j=0}^{m} T_{\text{push\_back}} + T_{\text{sort}} + T_{\text{retorno}}$$

Onde:

- n: número de elementos em `charFrequency`
- m: número de elementos em `stringFrequency`
- $k = n + m$: número total de elementos

Desse modo, sabendo que:

- $T_{\text{inicialização}} = O(1)$ para inicialização de vetor vazio.
- $T_{\text{push\_back}} = O(1)$ que é o tempo amortizado para inserção no final do vetor [11].
- $T_{\text{sort}} = O(k \log k)$ para ordenação com `std::sort` [37].
- $T_{\text{retorno}} = O(k)$ já que o retorno realiza uma cópia do vetor.
- $T_{\text{sortByFrequency}} = O(1)$ complexidade para a função de comparação constante assumida.

Podemos afirmar então que a complexidade total da função é:

$$T_{\texttt{unifyAndSort}} = O(1) + O(n) + O(m) + O(k \log k) + O(k) = O(k \log k)$$

Considerando que $k = n + m$, temos:

$$T_{\texttt{unifyAndSort}} = O((n + m) \log(n + m))$$

5.1.4 Função `printToFile()` - Escrita de resultados em arquivo

A função `printToFile()` grava os tokens ordenados com suas frequências em um arquivo de texto.

Parâmetros

`string filename` Nome do arquivo de saída, onde a tabela de frequência será gravada.

Listing 8: printToFile() - main.cpp

```
void printToFile(string filename){
    ofstream outputFile(outputPath + filename + ".txt");
    for (pair<string, int> i : sortedTokens){
```

```
4           outputFile << '"' << i.first << '"' << ' ' << i.second << endl;
5       }
6   }
```

Podemos formalizar a análise utilizando o **Teorema da Adição**:

$$T_{\texttt{printToFile}} = T_{\text{concatenação}} + T_{\text{abertura}} + \sum_{i=0}^{k} T_{\text{escrita}}$$

Onde:

- k: número de elementos em `sortedTokens`
- l_i: comprimento da string do i-ésimo token
- d_i: número de dígitos do valor inteiro do i-ésimo token

Desse modo, sabendo que:

- $T_{\text{concatenação}} = O(p+q+r)$ onde p, q e r são os comprimentos de `outputPath`, `filename` e `".txt"` respectivamente [38];
- $T_{\text{abertura}} = O(1)$ já que abertura de arquivo é operação de E/S constante [39];
- $T_{\text{escrita}} = O(l_i + d_i)$ para cada token, considerando a escrita da string entre aspas e o valor numérico [40].

Podemos afirmar então que a complexidade total da função é:

$$T_{\texttt{printToFile}} = O(p+q+r) + O(1) + \sum_{i=0}^{k} O(l_i + d_i)$$

Considerando que:

- $p+q+r$ são constantes na prática já que caminhos de arquivo têm comprimento limitado;
- $d_i = O(\log_{10} f_i)$ onde f_i é a frequência do token;
- $\sum_{i=0}^{k} l_i = L$ onde L é o comprimento total de todas as strings.

Temos:

$$T_{\texttt{printToFile}} = O(1) + O(L + k \cdot \log_{10} f_{\max})$$

Onde $f_{\max}$ é a frequência máxima encontrada.
E ao simplificarmos, acabamos com $O(k)$, já que, como discutido anteriormente, L é basicamente uma constante.

5.1.5 Complexidade Total da Main

Considerando que a função `main` tem suporte a processamento de diretórios, a complexidade total do sistema pode ser expressa como:

$$T_{\text{sistema}} = O(1) + O(1) + \sum_{i=1}^{F} O(n_i) + O(k \log k) + O(k)$$

Onde:

- F = número total de arquivos processados

- n_i = tamanho do i-ésimo arquivo
- k = número total de símbolos únicos acumulados de todos os arquivos
- $O(1)$ = complexidade de `getFullCharMap`
- $\sum_{i=1}^{F} O(n_i)$ = soma das complexidades de processamento de todos os arquivos
- $O(k \log k)$ = complexidade de `unifyAndSort`
- $O(k)$ = complexidade de `printToFile`

No pior caso, onde todos os arquivos são processados:

$$\boxed{T_{\text{sistema}} = O\left(\sum_{i=1}^{F} n_i + k \log k\right)}$$

5.2 Compressor e Descompressor Huffman

A função `main` do sistema de compressão Huffman atua como ponto de entrada do programa, implementando um sistema completo de compressão e descompressão baseado no algoritmo de Huffman. O programa opera em dois modos principais: compressão e descompressão de arquivos.

Parâmetros

`argc` : Número de argumentos da linha de comando.
`argv` : Vetor contendo os argumentos passados ao programa.

Retorno

Código de retorno (0 para sucesso, 1 para erro)

Listing 9: main.cpp - Sistema de Compressão Huffman

```
int main (int argc, char* argv[]){
    string argv1, argv2, argv3;
    if (argc >= 2) argv1 = argv[1];
    if (argc >= 3) argv2 = argv[2];
    if (argc >= 4) argv3 = argv[3];

    if (argc == 2 && (argv1 == "--help" || argv1 == "-h")) {
        cout << "Usage: " << argv[0] << " <frequency_sheet> <file_to_compress>"
             << endl;
        return 0;
    }

    cout << "Arguments received: " << endl;

    if (argv1 == "--compress" || argv1 == "-c") {
        cout << "Compressing file: " << argv3 << " using frequency sheet: " <<
            argv2 << endl;
        processFile(argv2); //popula o unordered_map frequencySheet
        buildHuffmanTree(frequencySheet);
        compressFile(argv3);
        return 0;
    }

    if (argv1 == "--decompress" || argv1 == "-d") {
        cout << "Decompressing file: " << argv3 << " using frequency sheet: "
            << argv2 << endl;
        processFile(argv2); //popula o unordered_map frequencySheet
```

```
25          buildHuffmanTree(frequencySheet);
26          decompressFile(argv3);
27          return 0;
28      }
29      return 0;
30  }
```

Ao analisar a função `main`, identifica-se uma estrutura de controle baseada em argumentos que direciona a execução para diferentes fluxos funcionais. Para compreender o desempenho total do sistema, é necessário analisar a complexidade de cada uma das funções auxiliares individualmente, considerando os diferentes caminhos de execução.

A análise deve aplicar o **Teorema da Adição** para operações sequenciais, porém com a particularidade de que as funções são executadas em diferentes combinações dependendo do modo de operação escolhido.

Podemos formalizar a complexidade considerando os dois fluxos principais:

- **Modo de Compressão (-c/–compress)**: $T_{\text{compressão}} = T_{\text{processFile}} + T_{\text{buildHuffmanTree}} + T_{\text{compressFile}}$
- **Modo de Descompressão (-d/–decompress)**:$T_{\text{descompressão}} = T_{\text{processFile}} + T_{\text{buildHuffmanTree}} + T_{\text{decompressFile}}$

5.2.1 Análise das Operações de Controle

As operações iniciais de processamento de argumentos possuem complexidade constante:

$$T_{\text{processamento argumentos}} = \sum_{i=1}^{3} T_{\text{cópia string}} + T_{\text{verificação help}} + T_{\text{verificação modo}}$$

Onde:

- $T_{\text{cópia string}} = O(l_i)$ - cópia de cada argumento, onde l_i é o comprimento do i-ésimo argumento
- $T_{\text{verificação help}} = O(1)$ - comparações de string com valores constantes
- $T_{\text{verificação modo}} = O(1)$ - operações de comparação

Considerando que o comprimento máximo de argumentos é limitado por constantes do sistema:

$$T_{\text{processamento argumentos}} = O(1)$$

5.2.2 Função `processFile()` - Processamento de arquivo de frequências

A função `processFile()` tem como objetivo processar um arquivo de texto contendo pares chave-valor no formato específico, onde cada linha representa um símbolo e sua frequência correspondente, populando assim a estrutura `frequencySheet` para posterior construção da árvore de Huffman.

Listing 10: processFile() - main.cpp

```
1  void processFile(const string& filename){
2      ifstream file(filename);
3      if(!file.is_open()){
4          cerr << "Error: unable to open '" << filename << "'." << endl;
5          return;
6      }
7      regex linePattern(R"REG(^"((?:\\x[0-9A-Fa-f]{2}|\\[abfnrtv\\'"]|[A-Za-z0-9_
           ]|.)+)" ([0-9]+)$)REG");
8      string line;
9      smatch match;
```

```
10
11      while (getline(file, line)) {
12          if (regex_match(line, match, linePattern)) {
13              string key = match[1].str();   // conteúdo entre aspas
14              int value = stoi(match[2].str()); // número após o espaço
15              frequencySheet.insert({key, value});
16          } else {
17              cerr << "Linha inválida: " << line << endl;
18          }
19      }
20  }
```

Podemos formalizar a análise utilizando o **Teorema da Adição** e o **Teorema da Multiplicação**:

$$T_{\texttt{processFile}} = T_{\text{abertura arquivo}} + T_{\text{compilação regex}} + \sum_{i=1}^{L} \left(T_{\text{getline}} + T_{\text{regex\_match}} + T_{\text{extração dados}} + T_{\text{inserção mapa}} \right)$$

Desse modo, sabendo que:

- A abertura de arquivo em C++ é uma operação de complexidade constante **O(1)** em relação ao tamanho do conteúdo, dependendo apenas do sistema de arquivos.

$$T_{\text{abertura arquivo}} = O(1)$$

- A compilação de expressões regulares em C++ tem complexidade **O(p)** onde p é o comprimento do padrão. Considerando o padrão como constante:

$$T_{\text{compilação regex}} = O(1)$$

- A operação `getline` tem complexidade **O(l)** onde l é o comprimento da linha lida [**?**].
- A operação `regex_match` tem complexidade que pode variar, mas no caso médio é **O(l)** onde l é o comprimento da string de entrada, embora no pior caso possa chegar a $\mathbf{O(2^l)}$ para padrões complexos [**?**].
- A extração de dados via `match.str()` e `stoi()` são operações de complexidade **O(l)** e **O(d)** respectivamente, onde d é o número de dígitos no valor numérico.

$$T_{\text{extração dados}} = O(l) + O(d) = O(l)$$

- A inserção no `unordered_map frequencySheet` tem complexidade média **O(1)** e pior caso **O(n)** onde n é o número de elementos no mapa.

Considerando L como o número total de linhas no arquivo e l_{max} como o comprimento máximo de uma linha, podemos afirmar então que a complexidade total da função é:

$$T_{\texttt{processFile}} = O(1) + O(1) + \sum_{i=1}^{L} \left(O(l_i) + O(l_i) + O(l_i) + O(1) \right)$$

Simplificando:

$$T_{\texttt{processFile}} = O(1) + \sum_{i=1}^{L} O(l_i)$$

Como a soma dos comprimentos de todas as linhas é igual ao tamanho total do arquivo N:

$$\sum_{i=1}^{L} l_i = N$$

Portanto, a complexidade final é:

$$T_{\texttt{processFile}} = O(N)$$

5.2.3 Função `storeCodes()` - Armazenamento dos Códigos Huffman

A função `storeCodes()` tem como objetivo percorrer a árvore de Huffman recursivamente e armazenar os códigos binários correspondentes a cada símbolo folha na estrutura `huffmanCodes`.

Listing 11: storeCodes() - huffman.cpp

```
void storeCodes(const NodePtr& root, const string& str){
    if (!root) return;

    if (!root->left && !root->right){
        huffmanCodes[root->data] = str;
        return;
    }

    storeCodes(root->left, str + "0");
    storeCodes(root->right, str + "1");
}
```

Podemos formalizar a análise utilizando o **Teorema da Adição** para operações recursivas:

$$T_{\texttt{storeCodes}} = T_{\text{verificação nulo}} + T_{\text{verificação folha}} + T_{\text{armazenamento código}} + T_{\texttt{storeCodes}_{\text{esquerda}}} + T_{\texttt{storeCodes}_{\text{direita}}}$$

Desse modo, sabendo que:

- $T_{\text{verificação nulo}} = O(1)$ - operação de verificação de ponteiro nulo;
- $T_{\text{verificação folha}} = O(1)$ - verificação se o nó é folha;
- $T_{\text{armazenamento código}} = O(1)$ - inserção no mapa `huffmanCodes`;
- $T_{\text{concatenação string}} = O(l)$ - onde l é o comprimento atual do código;

Considerando que a árvore de Huffman tem k folhas (símbolos únicos) e $2k-1$ nós no total, e que o comprimento máximo do código é $L_{\max}$, a complexidade total é:

$$T_{\texttt{storeCodes}} = O(k \cdot L_{\max})$$

5.2.4 Função `updateMaxTokenLength()` - Atualização do Comprimento Máximo

A função `updateMaxTokenLength()` calcula o comprimento máximo entre todos os tokens armazenados nos códigos Huffman.

Listing 12: updateMaxTokenLength() - huffman.cpp

```
void updateMaxTokenLength(){
    for (const auto pair : huffmanCodes){
        if(pair.first.length() > maxTokenLength){
            maxTokenLength = pair.first.length();
        }
    }
}
```

A análise de complexidade é direta:

$$T_{\texttt{updateMaxTokenLength}} = \sum_{i=1}^{k} \left(T_{\text{acesso comprimento}} + T_{\text{comparacao}} + T_{\text{atribuicao}} \right)$$

Onde:

- $T_{\text{acesso comprimento}} = O(1)$ - operação `string::length()`;
- $T_{\text{comparacao}} = O(1)$ - comparação entre inteiros;
- $T_{\text{atribuicao}} = O(1)$ - atribuição de valor;

Como todas as operações internas são $O(1)$ e o loop executa k vezes (número de códigos Huffman):

$$T_{\texttt{updateMaxTokenLength}} = O(k)$$

5.2.5 Função `buildHuffmanTree()` - Construção da Árvore de Huffman

A função `buildHuffmanTree()` implementa o algoritmo clássico de construção da árvore de Huffman a partir da tabela de frequências.

Listing 13: buildHuffmanTree() - huffman.cpp

```
void buildHuffmanTree(const unordered_map<string, size_t>& frequencySheet){
    MinHeap pq;

    for (const auto& pair : frequencySheet) {
        pq.push(make_shared<HuffmanNode>(escapeCharToChar(pair.first), pair.
            second));
    }

    while (pq.size() > 1) {
        auto left = pq.pop();
        auto right = pq.pop();

        size_t sum = left->frequency + right->frequency;
        pq.push(make_shared<HuffmanNode>(string(), sum, left, right));
    }

    treeRoot = pq.pop();
    storeCodes(treeRoot, "");
    updateMaxTokenLength();
}
```

Podemos decompor a análise em três fases principais:

$$T_{\texttt{buildHuffmanTree}} = T_{\text{inicialização heap}} + T_{\text{construção árvore}} + T_{\texttt{storeCodes}} + T_{\texttt{updateMaxTokenLength}}$$

Onde:

- $T_{\text{inicialização heap}} = O(k \log k)$ - inserção de k elementos no min-heap através de inserções sequenciais [16, 17];
- $T_{\text{construção árvore}} = O(k \log k)$ - $k - 1$ iterações com operações de pop e push no heap, seguindo o algoritmo de Huffman [20, 16];
- $T_{\texttt{storeCodes}} = O(k \cdot L_{\max})$ - como analisado anteriormente, onde $L_{\max}$ é o comprimento máximo do código Huffman

- $T_{\texttt{updateMaxTokenLength}} = O(k)$ - como analisado anteriormente, percorrendo todos os códigos uma vez;

Portanto, a complexidade total é:

$$T_{\texttt{buildHuffmanTree}} = O(k \log k) + O(k \log k) + O(k \cdot L_{\max}) + O(k) = O(k \cdot L_{\max})$$

5.2.6 Função `bitsToBytes()` - Conversão de Bits para Bytes

A função `bitsToBytes()` tem como objetivo converter uma string de bits em um vetor de bytes, agrupando os bits em bytes completos e tratando o padding final quando necessário.

Listing 14: bitsToBytes() - compression.cpp

```
vector<unsigned char> bitsToBytes(const string& bits) {
    vector<unsigned char> bytes;
    unsigned char currentByte = 0;
    int bitCount = 0;

    for (char bit : bits) {
        currentByte <<= 1;
        if (bit == '1') currentByte |= 1;
        bitCount++;

        if (bitCount == 8) {
            bytes.push_back(currentByte);
            currentByte = 0;
            bitCount = 0;
        }
    }

    if (bitCount > 0) {
        currentByte <<= (8 - bitCount);
        bytes.push_back(currentByte);
    }

    return bytes;
}
```

Podemos formalizar a análise utilizando o **Teorema da Adição**:

$$T_{\texttt{bitsToBytes}} = T_{\text{inicialização}} + \sum_{i=1}^{n} T_{\text{processamento bit}} + T_{\text{finalização}}$$

Onde:

- $T_{\text{inicialização}} = O(1)$ - criação do vetor e variáveis auxiliares
- $T_{\text{processamento bit}} = O(1)$ por bit - operações de shift e OR bit a bit [10]
- $T_{\text{push\_back}} = O(1)$ amortizado - inserção no vetor [11]
- $T_{\text{finalização}} = O(1)$ - tratamento do padding final

Considerando n como o número de bits na string de entrada:

$$T_{\texttt{bitsToBytes}} = O(1) + \sum_{i=1}^{n} O(1) + O(1) = O(n)$$

5.2.7 Função `findLongestToken()` - Busca do Token Mais Longo

A função `findLongestToken()` busca pelo token mais longo possível no arquivo que possua um código Huffman correspondente.

Listing 15: findLongestToken() - compression.cpp

```
string findLongestToken(ifstream& file){
    string longestMatch;
    string currentMatch;
    char ch;
    auto startPos = file.tellg();

    for(size_t i = 0; i < maxTokenLength && file.get(ch); i++){
        currentMatch += ch;
        if(huffmanCodes.find(currentMatch) != huffmanCodes.end()){
            longestMatch = currentMatch;
        }
    }

    file.clear();
    if (startPos != -1) {
        file.seekg(startPos + static_cast<std::streamoff>(longestMatch.length()
            ));
    }
    return longestMatch;
}
```

$$T_{\texttt{findLongestToken}} = T_{\text{posição arquivo}} + \sum_{i=1}^{L} T_{\text{leitura caractere}} + T_{\text{busca mapa}} + T_{\text{reposicionamento}}$$

Onde:

- $T_{\text{posição arquivo}} = O(1)$ - operação `tellg()` [12]
- $T_{\text{leitura caractere}} = O(1)$ - operação `file.get(ch)`
- $T_{\text{concatenação string}} = O(1)$ amortizado - operação `+=` com caractere [38]
- $T_{\text{busca mapa}} = O(L)$ no pior caso - busca em `unordered_map` pode degradar com colisões [14]
- $T_{\text{reposicionamento}} = O(1)$ - operações `seekg()` [15]
- $L = \min(\text{maxTokenLength}, \text{caracteres disponíveis})$

$$T_{\texttt{findLongestToken}} = O(1) + \sum_{i=1}^{L}(O(1) + O(L)) + O(1) = O(L^2)$$

5.2.8 Função `compressFile()` - Compressão do Arquivo

A função `compressFile()` coordena todo o processo de compressão, desde a leitura do arquivo original até a escrita do arquivo comprimido.

Listing 16: compressFile() - compression.cpp

```
void compressFile(const string& filename){
    ifstream inFile(filename);
```

```
    ofstream outFile(filename + "_compressed.bin", ios::binary);
    string allBits = "";
    if (!inFile.is_open()) {
        cerr << "Error: unable to open '" << filename << "'." << endl;
        return;
    }
    if (!outFile.is_open()){
        cerr << "Error: unable to create compressed file." << endl;
        return;
    }

    while(inFile.peek() != EOF){
        string token = findLongestToken(inFile);
        if(!token.empty()){
            allBits += huffmanCodes.at(token);
        } else {
            char ch;
            inFile.get(ch);
            cerr << "Error: character not found in FrequencySheet " << ch <<
                endl;
            return;
        }
    }

    int padding = (8 - (allBits.length()%8)) % 8;
    string binPadding = decToBinary(padding); //quantos bits faltam para que
        tudo seja multiplo de 8

    allBits.append(padding, '0');

    unsigned char paddingByte = static_cast<unsigned char>(padding);

    outFile.write(reinterpret_cast<const char*>(&paddingByte), 1);

    for(size_t i = 0; i < allBits.size(); i += 8){
        string byteStr = allBits.substr(i, 8);
        unsigned char byte = static_cast<unsigned char>(stoi(byteStr, nullptr,
            2));
        outFile.write(reinterpret_cast<const char*>(&byte), 1);
    }
    inFile.close(); outFile.close();
    cout << "Succesfull compression!" << endl;
}
```

$$T_{\texttt{compressFile}} = T_{\text{abertura arquivos}} + \sum_{i=1}^{T} T_{\texttt{findLongestToken}} + T_{\text{concatenação bits}} + T_{\text{cálculo padding}} + T_{\text{escrita arquivo}}$$

Onde:

- $T_{\text{abertura arquivos}} = O(1)$ - operações de abertura de arquivo
- T = número total de tokens no arquivo
- $T_{\texttt{findLongestToken}} = O(L^2)$ - como analisado anteriormente
- $T_{\text{concatenação bits}} = O(c_i)$ - onde c_i é o comprimento do código Huffman do i-ésimo token

- $T_{\text{cálculo padding}} = O(1)$ - operações matemáticas simples
- $T_{\text{escrita arquivo}} = O(B)$ - onde B é o número total de bytes escritos

Considerando que a soma dos comprimentos de todos os códigos Huffman é igual ao número total de bits N_{bits}:

$$T_{\texttt{compressFile}} = O(1) + \sum_{i=1}^{T} O(L^2) + O(N_{\text{bits}}) + O(1) + O(B)$$

Simplificando e considerando que $B = O(N_{\text{bits}})$ e $T \cdot L^2$ domina:

$$T_{\texttt{compressFile}} = O(T \cdot L^2 + N_{\text{bits}})$$

5.2.9 Função `decodeBitSequence()` - Decodificação de Sequência de Bits

A função `decodeBitSequence()` tem como objetivo decodificar uma sequência de bits percorrendo a árvore de Huffman até encontrar um símbolo folha correspondente, navegando pela árvore baseada nos bits de entrada.

Parâmetros

`root`: Nó raiz da árvore de Huffman.
`bits`: String contendo a sequência de bits a ser decodificada.
`index`: Índice atual na string de bits (passado por referência).

Retorno

`string`: Símbolo decodificado correspondente à sequência de bits.

Listing 17: decodeBitSequence() - decompression.cpp

```
string decodeBitSequence(const NodePtr root, const string& bits, size_t& index)
    {
    NodePtr current = root;

    while (!current->isLeaf() && index < bits.size()) {
        if (bits[index] == '0') {
            current = current->left;
        } else {
            current = current->right;
        }
        ++index;
        if (!current) break;
    }

    if (current && current->isLeaf()) {
        return current->data;
    } else {
        cerr << "Error: could not get to a leaf node in the Huffman Tree." <<
            endl;
        return "";
    }
}
```

Podemos formalizar a análise utilizando o **Teorema da Adição** e o **Teorema da Multiplicação**:

$$T_{\texttt{decodeBitSequence}} = T_{\text{inicialização}} + \sum_{i=0}^{D} \left(T_{\text{verificação folha}} + T_{\text{acesso bit}} + T_{\text{navegação árvore}} + T_{\text{incremento índice}} + T_{\text{verificação nulo}} \right) + T_{\text{}}$$

Onde:

- $T_{\text{inicialização}} = O(1)$ - atribuição do nó raiz à variável current
- $T_{\text{verificação folha}} = O(1)$ - operação `isLeaf()` no nó [1]
- $T_{\text{acesso bit}} = O(1)$ - acesso a elemento específico da string usando operator[] [2]
- $T_{\text{navegação árvore}} = O(1)$ - seguimento de ponteiro left ou right
- $T_{\text{incremento índice}} = O(1)$ - operação de incremento do índice
- $T_{\text{verificação nulo}} = O(1)$ - verificação se o ponteiro current é nulo
- $T_{\text{retorno símbolo}} = O(1)$ - retorno da string do nó folha
- D = profundidade máxima percorrida na árvore (limitada por $L_{\max}$, o comprimento máximo do código Huffman)

  Considerando que no pior caso a função percorre até $L_{\max}$ bits para decodificar um símbolo:

$$T_{\texttt{decodeBitSequence}} = O(1) + \sum_{i=0}^{L_{\max}} \big(O(1) + O(1) + O(1) + O(1) + O(1)\big) + O(1) = O(L_{\max})$$

5.2.10 Função `decompressFile()` - Descompressão do Arquivo

A função `decompressFile()` coordena todo o processo de descompressão, desde a leitura do arquivo comprimido até a reconstrução do arquivo original, incluindo o tratamento do padding e a escrita do resultado.

Parâmetros

`filename`: Nome do arquivo comprimido a ser descomprimido.

Listing 18: decompressFile() - decompression.cpp

```
void decompressFile(const string& filename){
    string allBits = binToString(filename);

    if (allBits.empty()) {
        cerr << "Error: no bits read from file '" << filename << "'" << endl;
        return;
    }

    if (allBits.size() < 8) {
        cerr << "Error: input too short to contain padding info" << endl;
        return;
    }

    string binPadding = allBits.substr(0, 8);
    int padding = stoi(binPadding, nullptr, 2);

    string dataBits = allBits.substr(8);
    if (padding > 0 && padding <= static_cast<int>(dataBits.size())) {
        dataBits = dataBits.substr(0, dataBits.size() - padding);
    }

    string finalText = "";
    size_t index = 0;
    while (index < dataBits.length()) {
        string decoded = decodeBitSequence(treeRoot, dataBits, index);
        if (decoded.empty()) break;
```

```
27          finalText.append(decoded);
28      }
29
30      size_t pos = filename.find("_compressed.bin");
31      string basename = filename;
32      if (pos != string::npos) {
33          basename = filename.substr(0, pos);
34      }
35
36      string outFilename = basename;
37      string extension = ".cpp";
38      ofstream outFile(outFilename + "_decompressed" + extension);
39      if (!outFile) {
40          cerr << "Error: could not create output file '" << outFilename << "'"
                << endl;
41          return;
42      }
43
44      outFile << finalText;
45      outFile.close();
46  }
```

Podemos decompor a análise em fases principais utilizando o **Teorema da Adição**:

$$\begin{aligned} T_{\texttt{decompressFile}} =& T_{\text{leitura bits}} + T_{\text{verificações iniciais}} + T_{\text{processamento cabeçalho}} \\ &+ \sum_{i=1}^{S} T_{\texttt{decodeBitSequence}_i} + T_{\text{processamento nome arquivo}} + T_{\text{escrita arquivo}} \end{aligned} \tag{1}$$

Onde:

- $T_{\text{leitura bits}} = O(B)$ - leitura do arquivo binário através de `binToString()`, onde B é o número de bytes no arquivo comprimido
- $T_{\text{verificações iniciais}} = O(1)$ - verificações de empty() e size() [3][4]
- $T_{\text{processamento cabeçalho}} = T_{\text{substr}} + T_{\text{stoi}} + T_{\text{remoção padding}} = O(1) + O(1) + O(1) = O(1)$ - operações com strings de tamanho fixo [5][6]
- $S =$ número de símbolos decodificados no arquivo
- $T_{\texttt{decodeBitSequence}_i} = O(L_{\max})$ - como analisado anteriormente para cada símbolo
- $T_{\text{concatenação}} = O(l_i)$ - onde l_i é o comprimento do i-ésimo símbolo decodificado [38]
- $T_{\text{processamento nome arquivo}} = T_{\text{find}} + T_{\text{substr}} = O(F) + O(F) = O(F)$ - onde F é o comprimento do nome do arquivo [8][5]
- $T_{\text{escrita arquivo}} = O(N)$ - escrita do texto final, onde N é o tamanho do arquivo original descomprimido [9]

Considerando que a soma dos comprimentos de todos os símbolos é igual ao tamanho original N e que o número de símbolos S está relacionado com N e os comprimentos médios dos códigos:

$$T_{\texttt{decompressFile}} = O(B) + O(1) + O(1) + \sum_{i=1}^{S} \big(O(L_{\max}) + O(l_i)\big) + O(F) + O(N)$$

Simplificando e considerando que $\sum_{i=1}^{S} l_i = N$ e que $S \cdot L_{\max}$ e N dominam:

$$T_{\texttt{decompressFile}} = O(S \cdot L_{\max} + N + B)$$

5.2.11 Relação entre Variáveis

- S $\frac{N}{l_{\text{avg}}}$ onde l_{avg} é o comprimento médio dos símbolos
- B $\frac{N \cdot L_{\text{avg}}}{8}$ onde L_{avg} é o comprimento médio dos códigos Huffman
- No caso típico, $S \cdot L_{\max} = O(N)$ e $B = O(N)$

Portanto, a complexidade final simplificada é:

$$T_{\texttt{decompressFile}} = O(N)$$

Onde N é o tamanho do arquivo original descomprimido.

5.2.12 Complexidade Consolidada do Sistema

Para o modo de compressão:

$$T_{\text{sistema}} = O(T \cdot L^2 + N_{\text{bits}})$$

Para o modo de descompressão:

$$T_{\text{sistema}} = O(n)$$

Onde:

- n: tamanho do arquivo de frequência
- T: número total de tokens no arquivo a comprimir
- L: comprimento máximo de um token
- N_{bits}: número total de bits na saída comprimida

Logo, pode-se ver que o termo dominante na compressão é $O(T \cdot L^2)$, que na prática pode ser simplificado considerando que:

- $T = O(n)$ (número de tokens proporcional ao tamanho do arquivo);
- L é limitado na prática (tokens têm comprimento máximo finito);
- L^2 pode ser considerado constante para análise assintótica.

Portanto, a complexidade final do sistema é:

$$\boxed{T_{\text{sistema}} = O(n)}$$

5.2.13 Considerações Finais

- **Eficiência Geral**: O sistema apresenta complexidade linear em relação ao tamanho da entrada, sendo eficiente para arquivos de diferentes portes
- **Bottleneck Identificado**: A função `findLongestToken` com complexidade $O(L^2)$ representa um potencial gargalo para tokens muito longos
- **Otimização Futura**: A substituição do algoritmo de busca do token mais longo por uma estrutura de dados mais eficiente (como uma trie) poderia reduzir a complexidade para $O(L)$
- **Desempenho Prático**: Na prática, o sistema é viável para a maioria dos casos de uso, com a compressão sendo a operação mais custosa

6 Resultados Experimentais

6.1 Desempenho de Compressão

Os testes realizados com o sistema de compressão Huffman demonstraram resultados significativos na redução de tamanho de arquivos de código-fonte C++:

Tabela 1: Resultados de Compressão por Arquivo

Arquivo Original	**Arquivo Comprimido**	**Redução**	**Taxa de Compressão**
50 KB	20 KB	30 KB	60%
1300 bytes	600 bytes	700 bytes	53.8%
12 KB	5 KB	7 KB	58,33%

6.2 Análise do Desempenho

Ao utilizar o contador de frequência treinado em um repositório grande de C++, observamos:

- Arquivos comprimidos para aproximadamente **45%** do tamanho original.
- Redução média de **55%** no tamanho dos arquivos.

6.3 Observações Técnicas

- A taxa de compressão varia significativamente dependendo de onde a tabela de frequências foi treinada;
- Tabelas treinadas em bases maiores e mais representativas tendem a oferecer melhores taxas de compressão;
- É necessário considerar o tamanho do arquivo .txt da tabela de frequências no cálculo do ganho total;
- Arquivos similares ao conjunto de treinamento comprimem melhor do que arquivos com padrões diferentes;
- O sistema não aceita caracteres com acentuação, o que pode limitar a compressão em arquivos com comentários em português;
- Arquivos menores apresentam taxas de compressão mais variáveis;
- A eficiência depende da similaridade entre o arquivo sendo comprimido e o conjunto usado para treinar a tabela de frequências;

7 Bibliografía

[1] CPPRET (CPPREFERENCE). *Member functions (cppreference)*. Disponível em: `https://en.cppreference.com/w/cpp/language/member_functions`. Acesso em: out. 2025.

[2] CPPRET (CPPREFERENCE). *std::string::operator[] (cppreference)*. Disponível em: `https://en.cppreference.com/w/cpp/string/basic_string/operator_at`. Acesso em: out. 2025.

[3] CPPRET (CPPREFERENCE). *std::string::empty (cppreference)*. Disponível em: `https://en.cppreference.com/w/cpp/string/basic_string/empty`. Acesso em: out. 2025.

[4] CPPRET (CPPREFERENCE). *std::string::size (cppreference)*. Disponível em: `https://en.cppreference.com/w/cpp/string/basic_string/size`. Acesso em: out. 2025.

[5] CPPRET (CPPREFERENCE). *std::string::substr (cppreference)*. Disponível em: `https://en.cppreference.com/w/cpp/string/basic_string/substr`. Acesso em: out. 2025.

[6] CPPRET (CPPREFERENCE). *std::stoi (cppreference)*. Disponível em: `https://en.cppreference.com/w/cpp/string/basic_string/stol`. Acesso em: out. 2025.

[7] CPPRET (CPPREFERENCE). *std::string::append (cppreference)*. Disponível em: `https://en.cppreference.com/w/cpp/string/basic_string/append`. Acesso em: out. 2025.

[8] CPPRET (CPPREFERENCE). *std::string::find (cppreference)*. Disponível em: `https://en.cppreference.com/w/cpp/string/basic_string/find`. Acesso em: out. 2025.

[9] CPPRET (CPPREFERENCE). *std::ostream::write (cppreference)*. Disponível em: `https://en.cppreference.com/w/cpp/io/basic_ostream/write`. Acesso em: out. 2025.

[10] C++ Reference, *Bitwise Operations*, 2023.

[11] C++ Reference, *std::vector::push_ back*, 2023.

[12] C++ Reference, *std::istream::tellg*, 2023.

[13] C++ Reference, *std::string::operator+=*, 2023.

[14] C++ Reference, *std::unordered_ map::find*, 2023.

[15] C++ Reference, *std::istream::seekg*, 2023.

[16] Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C., *Introduction to Algorithms*, 3ª edição, MIT Press, 2009.

[17] Cormen, T. H., et al., *Heap Operations*, Capítulo 6, Introduction to Algorithms, 2009.

[18] Huffman, D. A., *A Method for the Construction of Minimum-Redundancy Codes*, Proceedings of the IRE, 1952.

[19] Cover, T. M., Thomas, J. A., *Elements of Information Theory*, Wiley-Interscience, 1991.

[20] HUFFMAN, D. A. *A Method for the Construction of Minimum-Redundancy Codes*. Proceedings of the IRE, v. 40, n. 9, p. 1098–1101, 1952.

[21] SALOMON, D. *Data Compression: The Complete Reference*. 4th Edition, Springer, 2004.

[22] GEEKSFORGEEKS. *Priority Queue using Binary Heap*. Disponível em: `https://www.geeksforgeeks.org/dsa/priority-queue-using-binary-heap/`. Acesso em: out. 2025.

[23] GEEKSFORGEEKS. *map vs unordered_map in C++*. Disponível em: `https://www.geeksforgeeks.org/cpp/map-vs-unordered_map-c/`. Acesso em: out. 2025.

[24] CPPRET (CPPREFERENCE). *std::basic_ string (cppreference)*. Disponível em: `https://en.cppreference.com/w/cpp/string/basic_string/basic_string`. Acesso em: out. 2025.

[25] CPPRET (CPPREFERENCE). *std::vector (cppreference)*. Disponível em: `https://en.cppreference.com/w/cpp/container/vector/vector`. Acesso em: out. 2025.

[26] CPLUSPLUS. *std::string constructor — cplusplus.com*. Disponível em: `https://cplusplus.com/reference/string/string/string/`. Acesso em: out. 2025.

[27] CPPRET (CPPREFERENCE). *static_cast conversion (cppreference)*. Disponível em: https://en.cppreference.com/w/cpp/language/static_cast.html?utm_source=chatgpt.com. Acesso em: out. 2025.

[28] CPPRET (CPPREFERENCE). *std::basic_fstream (cppreference)*. Disponível em: https://en.cppreference.com/w/cpp/io/basic_fstream. Acesso em: out. 2025.

[29] GEEKSFORGEEKS. *How to Read Whole ASCII File Into C++ std::string?*. Disponível em: https://www.geeksforgeeks.org/cpp/how-to-read-whole-ascii-file-into-std-string-in-cpp/?utm_source=chatgpt.com. Acesso em: out. 2025.

[30] GEEKSFORGEEKS. *Different ways to access characters in a given String in C++*. Disponível em: https://www.geeksforgeeks.org/cpp/different-ways-to-access-characters-in-a-given-string-in-c/. Acesso em: out. 2025.

[31] KARANKEYASH. *Time-complexity in switch/case vs if/else*. Disponível em: https://karankeyash.hashnode.dev/time-complexity-in-switchcase-vs-ifelse. Acesso em: out. 2025.

[32] GEEKSFORGEEKS. *Converting Number to String in C++*. Disponível em: https://www.geeksforgeeks.org/cpp/converting-number-to-string-in-cpp/. Acesso em: out. 2025.

[33] CODE36O. *C++ String Concatenation*. Disponível em: https://www.naukri.com/code360/library/cpp-string-concatenation. Acesso em: out. 2025.

[34] GEEKSFORGEEKS. *std::string::length, std::string::capacity, std::string::size in C++ STL*. Disponível em: https://www.geeksforgeeks.org/cpp/std-string-length-std-string-capacity-std-string-size-in-cpp-stl/. Acesso em: out. 2025.

[35] GEEKSFORGEEKS. *std::string::length, std::string::capacity, std::string::size in C++ STL*. Disponível em: https://www.geeksforgeeks.org/cpp/std-string-length-std-string-capacity-std-string-size-in-cpp-stl/. Acesso em: out. 2025.

[36] SCALERTOPICS. *std::string::length, std::string::capacity, std::string::size in C++ STL*. Disponível em: https://www-scaler-com.translate.goog/topics/cpp-find/?_x_tr_sl=en&_x_tr_tl=pt&_x_tr_hl=pt&_x_tr_pto=tc&_x_tr_hist=true. Acesso em: out. 2025.

[37] SCALERTOPICS. *std::sort*. Disponível em: https://en.cppreference.com/w/cpp/algorithm/sort. Acesso em: out. 2025.

[38] C++ REFERENCE. *std::string::operator+*. Disponível em: https://en.cppreference.com/w/cpp/string/basic_string/operator%2B. Acesso em: out. 2025.

[39] C++ REFERENCE. *std::ofstream*. Disponível em: https://en.cppreference.com/w/cpp/io/basic_ofstream. Acesso em: out. 2025.

[40] C++ REFERENCE. *std::basic$_o$stream :: operator << .Disponvelem* : . Acesso em: out. 2025.

C++ REFERENCE. *std::basic$_o$stream :: $flush$.Disponvelem* : . Acesso em: out. 2025.