Algoritmo para compressão e descompressão de arquivos - LZW

Erik Roberto Reis Neves – 2022093040[1] and
Gabriel Campos Prudente – 2022069425[1]

[1]Departamento de ciência da computação, UFMG
erik.neves.ufmg@gmail.com
gabrielcp.ufmg@gmail.com

19 de novembro de 2024

Sumário

1 Introdução

Esta documentação tem o objetivo introduzir o algoritmo (implementado) de compressão e descompressão através do método ***Lempel-Ziv-Welch (LZW)***. Além disso, será abordada a implementação de uma árvore de prefixos (*do inglês, retrieval Trie*), que será fundamental para a implementação dessa ferramenta. Por fim, será disponibilizado uma análise do funcionamento do compressor e descompressor com diversos tipos de arquivos e suas métricas.

2 Métodologia

Esta seção apresenta uma visão geral do método LZW (Lempel-Ziv-Welch), incluindo uma análise detalhada das estruturas de dados utilizadas e a implementação das classes fundamentais que compõem esta ferramenta. Serão discutidas também as classes específicas para os processos de compressão e descompressão, destacando a funcionalidade e integração de cada uma. Entre as subseções, será explicado o papel essencial da Trie Compacta, que se mostra particularmente adequada à eficiência e à dinâmica do algoritmo LZW.

2.1 O método de compressão Lempel-Ziv-Welch (LZW)

O método de compressão LZW é uma técnica de compressão de dados sem perda, ou seja, comprimir e descomprimir não afeta a integridade dos dados originais. O princípio é a substituição de strings que se repetem muitas vezes ao longo de um arquivo, substituindo-as por um código (ou palavra-código), de modo a diminuir o número de bytes da saída. Para isso, o método se baseia na inserção de strings em um dicionário, que serão mapeadas para um código, que ficará no lugar dessa string no arquivo compactado.

A técnica de compressão recebe o arquivo e inicializa o dicionário com todos os caracteres do alfabeto de entrada (ASCII). Feito isso, o algoritmo itera sobre os símbolos concatenando-os e inserindo no dicionário, à medida que novas strings vão sendo formadas. A descompressão segue o processo inverso, pega-se cada código do arquivo compactado, verifica no dicionário qual string está associada a esse código e substitui. Logo, o método se comporta como uma fábrica de arquivos: recebe um arquivo normal, transforma-o em outro arquivo compactado e transforma novamente no arquivo normal.

O LZW será um *método* enquanto ainda não foi definido como armazenar as palavras no dicionário, nem o que é esse dicionário. Portanto, para torná-lo um

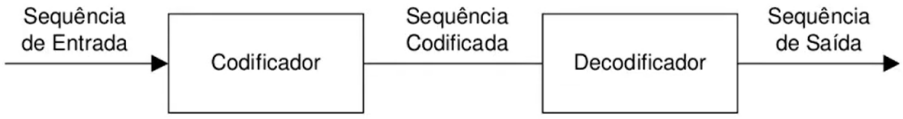

Figura 1: Figura ilustrativa do método de compressão LZW

algoritmo, é preciso responder uma pergunta central: **qual estrutura de dados eficiente que se comporte como um dicionário e que seja boa para armazenar grandes sequências e para a busca delas?** Para isso, iremos usar uma *Trie Compacta*, formalizando, nesse trabalho prático, o algoritmo de LZW.

2.2 Estruturas de Dados

A principal estrutura de dados utilizada é a **Árvore de Prefixos Compacta**, ou Trie compacta. Essa estrutura, é uma variação da **Trie** original, de forma que ela se adequa bem ao armazenar um grande volume de palavras, ou strings. Sua principal característica é a compactação dos nós que têm apenas um único filho, o que diminui a memória necessária para armazenar os dados, já que a condensação de nós permite que esses guardem substrings, ou prefixos de palavras, requerendo menos nós para a manutenção da estrutura da árvore. Além disso, essa implementação será usada como um dicionário, em que cada palavra inserida na árvore é uma chave, que é mapeada para um valor.

A razão do uso da trie compacta no método LZW é que ela se mostra um "dicionário" eficiente para lidar com inserções e buscas, especialmente quando há um volume grande de dados. Essa estrutura permite armazenar sequências comuns de forma compacta, economizando memória e acelerando as operações, pois os caminhos com prefixos iguais são unidos, reduzindo a profundidade da árvore, e consequentemente, o tempo de inserção e busca.

2.3 Trie Compacta

Essa subseção tem como objetivo introduzir o funcionamento das Trie Compacta, destacando seus componentes principais e principais métodos. Além disso, serão citadas, quando houver, decisões de projeto tomadas pelo grupo.

2.3.1 TrieNode

Essa classe representa um **nó** da *Trie Compacta*. Cada nó armazena as informações necessárias para a construção e correta utilização dessa estrutura de dados. Seus atributos são:

a) ***Previous***: Referência para o nó pai.

b) ***Substring***: Parte da chave associada ao nó, contendo um prefixo ou sequência específica de caracteres.

c) ***Value***: Valor associado ao nó, inicialmente *None*, mas definido caso o nó seja terminal, indicando o final de uma chave inserida (nó folha).

d) ***children***: Lista de referências para os nós filhos, configurada para até n filhos, onde n é o número de símbolos do alfabeto. Em uma Trie Compacta **binária** ($n = 2$) — usada neste trabalho — cada posição corresponde a um símbolo (0 ou 1), e há uma posição extra para indicar um nó especial que marca o final de uma chave completa, mesmo que o nó possua outros filhos. Os valores que serão armazenados na primeira posição do vetor *children*, isto é, *children[0]* sempre começam com 0, e os valores que armazenados na segunda posição do vetor, *children[1]* sempre começam com 1.

e) ***_isLeaf***: *Booleano* que indica se o nó é uma folha, atribuído quando *Value* possui um valor.

Além disso, a classe conta com métodos *Getters* e *Setters* para a *Substring* e o *Value* armazenados no nó.

2.3.2 Trie

Essa é a principal estrutura de dados utilizada. A **Trie** é uma combinação de *TrieNodes*, que são ligados através de arestas, que no código são representadas por referências. Ou seja, um *TrieNode* terá referência para outro *TrieNode*, formando assim as arestas. A Trie usada neste trabalho é a versão compacta, ou **Trie Compacta** (em inglês, *Compressed Trie*), em que os nós guardam prefixos ou substrings, em vez de armazenar apenas um símbolo. Como dito anteriormente, a principal característica da Trie é a aglutinação dos nós que têm apenas um único filho, fazendo com que a substring armazenada no nó filho seja concatenada com a substring do nó pai, permitindo que o nó possa armazenar uma substring inteira, em vez de apenas um símbolo. Seus atributos são:

a) ***Root***: A raiz da árvore, que, como decisão de projeto, será um *TrieNode* nulo, que não armazena nada, apenas a referência para os filhos.

b) ***NodeCount***: O número de *TrieNodes* (ou nós) na árvore Trie.

c) ***Depth***: A profundidade da árvore.

d) ***DetailedReturn***: Parâmetro opcional, definido por padrão como `False`, que indica se o retorno será mais detalhado, com informações de prefixos e sufixos, caso seja verdadeiro.

Quanto aos métodos, a Trie utilizada apresenta os principais métodos dessa estrutura de dados:

- **Insert**

```
1    def Insert(self, key, value):
```

Código fonte 1: *Trie.py* - Assinatura do método insert

Esse é o método que faz a inserção na trie compacta.

Parâmetros:

a) ***key***: A chave a ser inserida, que no caso, será uma string binária.

b) ***value***: O valor que será associado à chave, ou o valor para o qual a chave é mapeada.

Lógica da Implementação:

O algoritmo inicia na raiz e, em cada passo, pega o primeiro símbolo da substring atual da chave para definir o próximo nó na trie, guiando a inserção:

```
1    bit = int(key[i])
```

Código fonte 2: *Trie.py* - Pega o primeiro símbolo da substring atual

Após isso, o algoritmo prossegue enquanto não tiver feito uma varredura completa na chave. No processo, ele sempre verifica se há um nó criado em um filho especifico do nó atual:

```
    if currentNode.children[bit] is None:
        currentNode.children[bit] = _trieNode(currentNode, key, value)
        currentNode = currentNode.children[bit]
```

Código fonte 3: *Trie.py* - Verificação e criação de um novo nó filho da raiz

Caso esse nó já exista, o `if` é ignorado, e a inserção continua. A dinâmica da inserção é: enquanto não chegar ao fim da chave, o algoritmo compara símbolo a símbolo da chave com as substrings armazenadas nos nós já existentes. Sempre que há um símbolo da substring do nó atual que difere do símbolo atual da chave, é necessário criar nós para garantir a estabilidade da Trie. Para isso, realizamos manipulações na substring do nó e da chave: primeiro, guardamos o prefixo do nó atual, que corresponde aos símbolos que não diferiram da chave. Em seguida, novos nós são criados: um para armazenar o sufixo existente na substring do nó atual, e outro para armazenar o sufixo da chave, que é o símbolo que diferiu até o final da chave:

```
    if i + j >= len(key) or key[i + j] != substring[j]:
        prefixMatch = substring[:j]
        suffixExisting = substring[j:]
        suffixNew = key[i + j:]

        newChildExisting = _trieNode(childNode, suffixExisting, childNode.GetValue())
        [...]
        newChildNode = _trieNode(childNode, suffixNew, value)
```

Código fonte 4: *Trie.py* - Manipulação de prefixo e sufixo para criação de novos nós

Um ponto importante a se destacar é o uso de um nó extra para indicar se um nó é folha, mesmo se tiver filhos. Para isso, como destacado em **TrieNode**, em nós em que isso ocorre, eles irão ter um filho extra, com a variável `IS_WORD_END ( = "$")`:

```
if bit is not None and childNode.children[bit] is None and childNode.IsLeaf():
    newChildNode = _trieNode(currentNode, IS_WORD_FULL, childNode.GetValue())
    childNode.children[2] = newChildNode
    childNode.children[bit] = _trieNode(currentNode, key[i:], value)
```

Código fonte 5: *Trie.py* - Implementação de verificação para indicar nós folha

Retorno

Por padrão, será apenas a chave e o valor:

```
return (key, value) if not detailedReturn else (key, suffixNew, suffixExisting,
↪  newChildNode.GetValue())
```

Código fonte 6: *Trie.py* - Exemplo de retorno do método Insert

- **Search**

```
    def Search(self, key):
```

Código fonte 7: *Trie.py* - Assinatura do método Search

Esse método faz a busca de uma chave na Trie Compacta.

Parâmetros:

a) ***key***: Chave que será buscada.

Lógica da Implementação:

A busca inicia pela raiz. Ao iniciar, a busca começa verificando se há um nó correspondente para ir descer na árvore:

```
    child = currentNode.children[bit]

    if child is None:
        return None
```

Código fonte 8: *Trie.py* - Verificação de filho inexistente

Se o filho existir, a busca descerá para o nó correspondente e comparará a substring que está no nó com a porção correspondente da chave. Caso haja divergência de símbolos, a busca retornará *False*:

```
    if key[i:i + lenSubstring] != substring:
                return None
```

Código fonte 9: *Trie.py* - Comparação da substring da chave

Se não houverem diferenças de símbolos, o processo é repetido até que a chave de entrada seja lida completamente. Quando o nó em que a última correspondência foi feita for uma folha, a busca irá retornar *True*, indicando que a chave já foi inserida na Trie Compacta.

```
    return (key, suffix, currentNode._value, currentNode) if detailedReturn else
    ↪  currentNode._value
```

Código fonte 10: *Trie.py* - Exemplo de retorno da busca

Retorno

O retorno será `currentNode._value`, isto é, o valor da chave, caso a chave já tenha sido inserida na `Trie`, ou *None*, caso ela não tenha sido inserida.

```
1     return (key, suffix, currentNode._value, currentNode) if detailedReturn else
      ↪  currentNode._value
```

Código fonte 11: *Trie.py* - Exemplo de retorno do método Search

- **Remove**

```
1     def Search(self, key)
```

Código fonte 12: *Trie.py* - Assinatura do método Remove

Esse método faz a remoção de uma chave na Trie Compacta.

Parâmetros:

a) ***key***: Chave que será removida.

Lógica da Implementação:

O algoritmo inicia fazendo uma busca, usando o método `Search`, próprio da classe, com o *detailedReturn* setado para *True*, para conseguir acessar o nó atual da busca. Se o resultado da busca for *None*, então é retornado *False*, indicando que a chave não existe na árvore:

```
1     search_result = self.Search(key)
2     if search_result is None:
3             return False
```

Código fonte 13: *Trie.py* - Verificação da remoção quando a chave não existe na árvore

Caso a chave exista, o algoritmo prossegue. A ideia do algoritmo é fazer manipulações com o nó pai, chamado de ***parentNode***. Primeiro ele verifica se o nó atual na posição *SIGMA_SIZE* (posição extra que indica que um nó é folha, mesmo se ele tiver filhos). Caso seja, o nó pai já é o proprio nó atual, um vez que, por decisão de projeto, optamos por retornar o nó que guarda o final da chave, e não o nó que guarda a váriavel global `IS_WORD_END_FULL`, que seria o nó atual real. Se tal situação não ocorre, então o algoritmo sobe para seu pai, por meio de *previous*, que é uma referência(aresta) para o nó anterior ao atual na árvore:

```
1       if currentNode.children[SIGMA_SIZE] is None:
2           parentNode: _trieNode = currentNode.previous
3       else:
4           parentNode = currentNode
```

Código fonte 14: *Trie.py* - Definição do nó pai

Se o nó existe, então ele deixa de ser folha, e passa a ser um nó intermediário:

```
1   currentNode.SetValue(None)
```

Código fonte 15: *Trie.py* - Definição do nó pai

A remoção ainda verifica se o nó pai é uma raiz e se o nó atual possui filhos. Caso não, então o nó atual é removido complemente da árvore:

```
1   if parentNode is self.root and not any(child for child in currentNode.children):
2           parentNode.children[int(key[0])] = None
3           self._nodeCount -= 1
4           return (key, parentNode.GetSubstring(), value) if self.detailedReturn else True
```

Código fonte 16: *Trie.py* - Remoção de nó atual sem filho, e com pai

Após isso, é feita a verificação se o pai, após a remoção do nó atual, terá apenas um filho. Se sim, o nó filho é concatenado ao pai, e a busca retorna True. Se não, a busca apenas retorna True, pois ao fazer o nó atual deixar de ser folha na árvore, a remoção já está parcialmente feita, restando apenas a verificação citada. Além disso, é feito a verificação:

```
1   nonNoneChildrenCount = len([child for child in parentNode.children if child is not None])
2   if nonNoneChildrenCount - 1 == 1:
3       [...]
4       substringFusion = parentNode.children[bitFusion].substring
5       substringParentNode = parentNode.substring
6       substringParentNode += substringFusion
7       parentNode.SetSubstring(substringParentNode)
8   else:
9       for i, child in enumerate(currentNode.children):
10          if i != SIGMA_SIZE and child is not None:
11              child_to_fuse = child
12              break
13      if child_to_fuse is not None:
14          [...]
15          substringParentNode += substringFusion
16          parentNode.SetSubstring(substringParentNode)
17
```

Código fonte 17: *Trie.py* - Concatenção de nó pai e filho

Retorno

O retorno será *True*, caso a chave exista, e é folha, o que já é garantido na busca, ou *False*, caso ela não exista, ou não é folha

2.4 Gerenciamento de arquivos

Para auxiliar no gerenciamento de arquivos, foi implementada a classe `FileManager`, exibida no Código Fonte 18, que tem como objetivo fornecer os dados no formato correto para o uso do algoritmos de compressão e descompressão.

```python
class FileManager:
    @staticmethod
    def ReadFile(filePath):
        # implementação... (inclui tratamento de outros tipos internamente)

    @staticmethod
    def SaveTextFile(filePath, content):
        # implementação...

    @staticmethod
    def SaveBinaryFile(filePath, binaryString):
        # implementação...

    @staticmethod
    def ReadBinaryFile(filePath):
        # implementação...
```

Código fonte 18: *FileManager.py* - Gerenciador de arquivos

2.5 Conversões de tipos

A classe `BinaryConversor`, exibido no Código Fonte 19, é responsável por todas as conversões de dados entre diferentes formatos binários e textuais utilizados no algoritmo de compressão. Isso é essencial para que os dados sejam processados e armazenados corretamente.

```python
class BinaryConversor:
    @staticmethod
    def ConvertBinaryToString(binary):
        if not binary:
            return ""

        byte_data = bytes(int(binary[i:i + 8], 2) for i in range(0, len(binary), 8))
        return byte_data.decode('utf-8', errors='ignore')

    @staticmethod
    def ConvertIntegerToBinaryString(integer, size=0):
        return format(integer, f'0{size}b') if size else bin(integer)[2:]

    @staticmethod
    def ConvertPrefixToBinaryString(prefix):
        return ''.join(BinaryConversor.ConvertIntegerToBinaryString(ord(c)) for c in prefix)
```

Código fonte 19: *BinaryConversor.py* - Gerenciador de conversões

2.6 Compressor

A implementação apresentada na classe `LZWCompressor`, que pode ser visto no Código Fonte 20, utiliza uma Trie como estrutura de dados para armazenar os códigos gerados durante o processo de compressão. O algoritmo é inicializado com um conjunto de símbolos (ou alfabeto), e a compressão é realizada através da construção de um dicionário dinâmico à medida que os símbolos do conteúdo são processados.

```python
from BinaryConversor.BinaryConversor import BinaryConversor
from Trie.Trie import Trie

class LZWCompressor:
    def __init__(self, sigmaSize, controlBits, initialBitsSize, maxCodeBits, incrementableBits = False):
        self.dict = Trie()
        self.sigmaSize = sigmaSize
        self.controlBits = controlBits
        self.initialBitsSize = initialBitsSize
        self.maxCodeBits = maxCodeBits
        self.incrementableBits = incrementableBits

        # Inicializando o dicionário com as raízes iniciais
        for index in range(sigmaSize):
            bin_key = BinaryConversor.ConvertPrefixToBinaryString(chr(index))
            self.dict[bin_key] = index

    def Compress(self, content):
        # implementação...

    def __insertNewCode(self, prefix_with_char_key):
        # implementação...

    def __incrementCurrentCodeBits(self):
        # implementação...

    def __convertCodesToBinaryString(self, compressedList):
        # implementação...

    @staticmethod
    def ExtractCodeLenghtAndContent(input, controlBits):
        # implementação...

```

Código fonte 20: *LZWCompressor.py* - Compressor LZW

2.6.1 Inicialização do Algoritmo

O algoritmo começa com a criação de um dicionário (Trie) que contém as entradas iniciais para todos os símbolos possíveis do alfabeto, representados por seus códigos binários. O tamanho do alfabeto é especificado pelo parâmetro `sigmaSize`, e os primeiros códigos são atribuídos diretamente a esses símbolos, com base em sua posição no alfabeto. O parâmetro `initialBitsSize` define o número inicial de bits usados para codificar os símbolos, e o parâmetro `maxCodeBits` especifica o número máximo de bits que podem ser usados para os códigos durante o processo de compressão.

2.6.2 Processo de Compressão

Durante a compressão, o algoritmo lê cada símbolo do conteúdo e tenta expandir o prefixo atual (uma sequência de símbolos) para incluir o próximo símbolo. Se o prefixo concatenado com o próximo símbolo já estiver no dicionário, o algoritmo simplesmente o estende. Caso contrário, o código do prefixo anterior é adicionado à lista comprimida, e o novo prefixo é inserido no dicionário.

O dicionário é atualizado dinamicamente conforme novos prefixos são encontrados, e, sempre que o número de códigos atinge o limite máximo (`maxCode`), o número de bits utilizados para codificar os códigos é incrementado (se o parâmetro `incrementableBits` for `True`). Isso permite que o algoritmo se adapte à quantidade de dados a serem comprimidos, utilizando mais bits conforme necessário.

2.6.3 Codificação Binária

Ao final do processo de compressão, os códigos gerados são convertidos para uma representação binária compactada. O código mínimo necessário para representar os símbolos é calculado, e todos os códigos são convertidos para esse formato binário, gerando o conteúdo comprimido final. A sequência de controle, que indica o número de bits usados para os códigos, é inserida no início do conteúdo comprimido, seguida pelos códigos binários correspondentes.

2.7 Descompressor

A descompressão de um arquivo comprimido pelo algoritmo LZW é realizada pela classe `LZWDecompressor`, exibida no Código Fonte 21, que segue a lógica inversa da compressão, reconstruindo o dicionário dinâmico e mapeando os códigos binários de volta para suas representações de strings originais.

```python
from BinaryConversor.BinaryConversor import BinaryConversor
from Trie.Trie import Trie

class LZWDecompressor:
    def __init__(self, sigmaSize):
        self.dict = Trie()
        self.sigmaSize = sigmaSize

        # Inicializando o dicionário com as raízes iniciais
        for index in range(sigmaSize):
            bin_key = BinaryConversor.ConvertPrefixToBinaryString(chr(index))
            self.dict[bin_key] = chr(index)

    def Decompress(self, code_length, content):
        # implementação...

    def __getNextCode(self, content, index, code_length):
        # implementação...

```

Código fonte 21: *LZWDecompressor.py* - Descompressor LZW

2.7.1 Inicialização do Algoritmo

O processo de descompressão inicia-se com a leitura número de bits de controle, extraído do início da sequência comprimida, que indica o tamanho de bits que representam uma palavra no arquivo codificado. Com essa informação, o algoritmo pode determinar o número inicial de bits necessários para interpretar os códigos comprimidos. Em seguida, o dicionário é inicializado com as entradas correspondentes ao alfabeto original, representadas pelos códigos ASCII dos símbolos. A quantidade de símbolos iniciais é determinada pelo parâmetro `sigmaSize`, similar ao processo de compressão.

2.7.2 Processo de descompressão

A descompressão é realizada lendo-se os códigos comprimidos sequencialmente e reconstruindo as sequências de símbolos originais. O algoritmo utiliza uma abordagem baseada em prefixos para decodificar a sequência de forma eficiente, seguindo os seguintes passos:

1.) **Leitura do Primeiro Código**: O primeiro código é lido e traduzido diretamente para o símbolo correspondente no dicionário. Esse símbolo é imediatamente adicionado ao conteúdo descomprimido e é definido como o prefixo atual.

2.) **Iteração sobre os Códigos**: Para cada código subsequente, o algoritmo verifica se o código já está presente no dicionário:

   1.) **Caso esteja presente**: O algoritmo recupera a sequência de símbolos correspondente ao código, adiciona essa sequência ao conteúdo descomprimido e concatena o prefixo anterior com o primeiro símbolo da sequência decodificada. Esse novo prefixo é então adicionado ao dicionário com o próximo código disponível.

   2.) **Caso não esteja presente**: Este cenário ocorre quando o código faz referência a uma sequência ainda não inserida no dicionário. Nesse caso, o algoritmo infere que a sequência corresponde ao prefixo anterior concatenado com o seu primeiro símbolo, adicionando essa sequência ao conteúdo descomprimido e ao dicionário.

3.) **Finalização**: O processo continua até que todos os códigos comprimidos sejam processados. O resultado é o conteúdo descomprimido, que deve ser idêntico ao texto original antes da compressão.

3 Execução da aplicação

3.1 Execução do código Principal

O arquivo **src/main.py** é responsável por gerenciar tanto a execução dos módulos quanto as configurações padrão usadas. Os parâmetros usados são descritos abaixo, além de estarem disponpivel com mais detalhes na Imagem 2 Já as variáveis definidas podem ser visualizadas no Código Fonte 22.

1.) **max-code-bits**: Número máximo de bits para os códigos usados. Campo opcional.

2.) **analysis**: Seleção do método de análise. Campo opcional.

3.) **statistics**: Habilita a coleta de estatísticas e gráficos para a compressão e descompressão. Afeta o desempenho do algoritmo. Campo opcional.

4.) **operation**: Seleção da operação de compressão ou descompressão. Campo obrigatório.

5.) **origin**: Arquivo de origem. Campo obrigatório.

```
erik@Notebook:/mnt/d/UFMG-Storage/2024-1/trie-compressor$ python3 src/main.py -h
usage: main.py [-h] [--max-code-bits MAX_CODE_BITS] --operation {compress-fixed,compress-dynamic,decompress}
 [--analysis {None,cProfile,memray}] [--statistics STATISTICS]
              --origin ORIGIN --destiny DESTINY

Aplicação para compressão e descompressão de arquivos - LZW

options:
  -h, --help            show this help message and exit
  --max-code-bits MAX_CODE_BITS
                        Número máximo de bits para os códigos usados.
  --operation {compress-fixed,compress-dynamic,decompress}
                        Seleção da operação de compressão ou descompressão.
  --analysis {None,cProfile,memray}
                        Seleção do método de análise.
  --statistics STATISTICS
                        Habilita a geração de estatísticas na compressão e descompresão.
  --origin ORIGIN       Arquivo de origem.
  --destiny DESTINY     Arquivo de destino.
```

Figura 2: Parâmetros disponíveis na aplicação

6.) **destiny**: Arquivo de destino. Campo obrigatório.

```python
# Constantes
SIGMA_SIZE = 256                # Tamanho do alfabeto utilizado
DEFAULT_BITS = 12               # Tamanho padrao dos codigos
DINAMIC_BIT_SIZE_START_WITH = 9 # Tamanho inicial dos codigos no caso de numero incrementavel de bits
CODE_CONTROL_BITS = 32          # Tamanho do codigo de controle incluido no comeco do arquivo comprimido, usado na descompressao

```

Código fonte 22: *main.py* - Configurações da aplicação

3.2 Execução dos Testes

A pasta **tests** contém todos os testes feitos pelo grupo. Para executar, é necessário ter o `Pytest` instalado em sua máquina:

```
pip install pytest
```

Após isso, para rodar todos os testes, basta executar:

```
pytest
```

```
tests/test_trie.py::test_trie_remove_str_bin_not_exist[00000000] PASSED
tests/test_trie.py::test_trie_remove_str_bin_not_exist[11111111] PASSED
tests/test_trie.py::test_trie_remove_str_bin_not_exist[01010101] PASSED
tests/test_trie.py::test_trie_remove_str_bin_not_exist[00110011] PASSED
tests/test_trie.py::test_trie_remove_str_bin_not_exist[10011001] PASSED

============================== 88 passed in 8.71s ==============================
```

Figura 3: Terminal gerado pelo pytest

A execução será algo do tipo da Figura 3:

Além disso é possível verificar a cobertura de arquivos e linhas dos códigos de teste, executando:

```
pytest --cov --cov-report=term-missing --cov-report=html
```

4 Análise de Complexidade

Nesta seção, analisaremos as complexidades maior expressão no código, seja por alta complexidade, ou de importância para o trabalho prático. Além disso, nossa análise será feita por meio de análise amortizada[1] de complexidade, evitando exageros na análise.

4.1 Trie - Insert

A lógica da inserção é ler os simbolos da chave, um a um, até achar uma substring que não existe na arvore, e então inserí-la, fazendo as manipulações necessárias. No código implementado, há uma predominância do primeiro while, que itera simbolo à simbolo, na chave:

```
1     lenKey = len(key)
2     while i < lenKey:
3         [...]
```

Código fonte 23: *Trie.py* - Loop principal do método Insert

[1]Análise amortizada é um método para analisar a complexidade de tempo de um algoritmo. Diferente das outras análises que focam no tempo de execução no pior caso, a análise amortizada examina como um algoritmo irá se comportar na prática ou na média

Além disso, há outro loop, que itera na substring do nó atual:

```
1       j = 0
2       while j < lenSubstring and i + j < lenKey
3           [...]
```

Código fonte 24: *Trie.py* - Loop secundário do método Insert

Porém, ele não é expressivo, visto que ele será apenas do tamanho da substring do nó, chegando a ser no máximo `|key|`, em que *key* é a chave a ser inserida. Mesmo assim, ele não irá interferir na complexidade, visto que ao final, o `lenSubtring`, ou seja, o tamanho da substring do nó atual, é somado ao valor de i, que é o iterador da chave:

```
1       i += lenSubstring
```

Código fonte 25: *Trie.py* - Iterador da chave

Portanto, a complexidade será $O(|key|)$, em que o loop principal irá dominar o método de inserção, iterando símbolo a símbolo da chave, e a chave está quase toda inserida na árvore. Caso seja a primeira inserção, a complexidade se torna $\mathcal{O}(1)$, visto que não existe nó filho da raiz, e a chave é inserida completamente, sem que haja a iteração nela.

4.2 Trie - Search

A busca funciona de maneira semelhante a inserção. Ela itera na chave buscada e realiza comparações com as substrings dos nós da árvore. De maneira geraç, há um loop principal, que itera na chave, e outro interno, feito pela própria linguagem - Python - que compara um pedaço da chave com a string:

```
1       while i < lenKey:
2           [...]
3           if key[i:i + lenSubstring] != substring:
```

Código fonte 26: *Trie.py* - Loops principais da busca

Da mesma forma que na inserção, o tamanho das substrings atuais é somado ao iterador:

```
1       i += lenSubstring
```

Código fonte 27: *Trie.py* - Iterador da chave

Portanto, para o método `Search`, a busca também é $O(|key|)$, com o loop principal sendo o mais expressivo na busca.

4.3 Compressão e descompressão

Nesta subseção, analisamos a complexidade computacional da compressão e descompressão. No método de compressão, a Trie é utilizada para realizar inserções e buscas, cuja complexidade amortizada é $O(|key|)$, onde $|key|$ representa o tamanho da chave processada. Como o loop principal percorre todo o conteúdo de entrada, a complexidade total do método é $O(n \cdot |key|)$, sendo n o tamanho do conteúdo.

Por outro lado, o método de descompressão também utiliza a Trie para buscas de prefixos, com complexidade $O(|key|)$ por operação. Como essa busca é repetida para cada código lido no conteúdo comprimido, a complexidade total do processo é $O(m \cdot |key|)$, onde m é o número de códigos no conteúdo de entrada. A implementação foi projetada para reduzir reiterações desnecessárias e garantir eficiência através de operações amortizadas, mas o crescimento dinâmico da Trie e a manipulação de strings podem se tornar fatores limitantes em cenários com grandes volumes de dados.

5 Estratégias de Robustez

5.1 Testes do Software

Para garantir que o código funcione de maneira esperada, uma das práticas adotadas foi a verificação do código por meio de **testes**. Em geral, são *testes unitários* que fazem uma varredura quase que completa no código. O objetivo foi garantir que cada método tenha diversos testes, explorando minuciosamente seus casos de bordas e analisando sua corretude. Para isso, utilizamos o framework de testes **pytest**

Além disso, em alguns momentos, a equipe fez uso de uma abordagem de programação orientada a testes, visando obter maior clareza sobre os tipos de saídas esperadas e guiando o desenvolvimento para minimizar falhas.

```python
@pytest.mark.parametrize(
    "key, value, expected",
    [("000", 1, ("000", "000", None, 1)),
     ("111", 1, ("111", "111", None, 1)),
])
def test_trie_insert_str_bin_trivial_case(trie, key, value, expected):
    result = trie.Insert(key, value)
    assert result == expected
```

Código fonte 28: *test_trie.py* - Exemplo de teste

Os testes seguem diversos princípios de boas práticas de programação, destacando-se dois principais:

1. **Testes que verificam um comportamento, não métodos:**

```python
@pytest.mark.parametrize(
    "first_key, first_value, second_key, second_value, expected",
    [("00000", 1, "00011", 2, ("00011", "11", "00", 2)),
     ("101", 1, "100", 2, ("100", "0", "1", 2)),
     ("110", 1, "111", 2, ("111", "1", "0", 2)),
     ("0110", 1, "0111", 2, ("0111", "1", "0", 2)),
     ("1110", 1, "1111", 2, ("1111", "1", "0", 2)),
     ("00001", 1, "00000", 2, ("00000", "0", "1", 2)),
])
def test_trie_insert_two_str_bin_with_common_prefix(trie, first_key, first_value, second_key,
↪   second_value, expected):
    trie.Insert(first_key, first_value)
    result = trie.Insert(second_key, second_value)
    assert result == expected
```

Código fonte 29: *test_trie.py* - Testando o comportamento da inserção de duas chaves com prefixo em comum

2. **Testes com nomes legíveis e descritivos:**

```python
@pytest.mark.parametrize(
    "first_key, first_value, second_key, second_value, expected",
    [("000", 1, "111", 2, ("111", "111")),
])
def test_trie_search_with_two_str_bin_with_different_prefixes(trie, first_key, first_value,
↪   second_key, second_value, expected):
    trie.Insert(first_key, first_value)
    trie.Insert(second_key, second_value)
    search = trie.Search(second_key)
    assert search == expected
```

Código fonte 30: *test_trie.py* - Buscando chave após a inserção de duas strings com prefixo comum

3. **Testes de integridade dos dados após o uso das ferramentas de compressão e descompressão:**

```
def test_compressing_and_decompressing_content_with_line_break():
    compressor = LZWCompressor(SIGMA_SIZE, CODE_CONTROL_BITS, DEFAULT_CODE_BITS,
    ↪  DEFAULT_CODE_BITS)
    content = "Em Algoritmos II \n , falhar nos testes não é o fim"
    result = compressor.Compress(content)
    code_control_bits, compressed = LZWCompressor.ExtractCodeLenghtAndContent(result,
    ↪  CODE_CONTROL_BITS)

    decompressor = LZWDecompressor(SIGMA_SIZE)
    decompressedContent = decompressor.Decompress(code_control_bits, compressed)

    assert content == decompressedContent

def test_compressing_and_decompressing_only_dots():
    compressor = LZWCompressor(SIGMA_SIZE, CODE_CONTROL_BITS, DEFAULT_CODE_BITS,
    ↪  DEFAULT_CODE_BITS)
    content = ".........."
    result = compressor.Compress(content)
    code_control_bits, compressed = LZWCompressor.ExtractCodeLenghtAndContent(result,
    ↪  CODE_CONTROL_BITS)

    decompressor = LZWDecompressor(SIGMA_SIZE)
    decompressedContent = decompressor.Decompress(code_control_bits, compressed)

    assert content == decompressedContent
```

Código fonte 31: *test_lzw_decompressor.py* - Testes de compressão e descompressão

4. **Testes das dependências das classes principais:**

```python

# TESTES DA MANIPULAÇÃO DE BINÁRIOS
def test_convert_binary_to_string():
    converted = BinaryConversor.ConvertBinaryToString(CONTENT_BITS)
    assert converted == CONTENT

def test_convert_integer_to_binary_string():
    converted = BinaryConversor.ConvertIntegerToBinaryString(65, 8)
    assert converted == "01000001"

def test_convert_prefix_to_binary_string():
    converted = BinaryConversor.ConvertPrefixToBinaryString("ABC")
    assert converted == "100000110000101000011"

def test_convert_exclamation_prefix_to_binary():
    converted = BinaryConversor.ConvertPrefixToBinaryString("!")
    assert converted == '100001'

# TESTES DA MANIPULAÇÃO DE ARQUIVOS
def test_read_file_as_text():
    file_name = 'content.txt'
    create_temp_file (CONTENT, file_name)

    content_read = FileManager.ReadFile(file_name)
    remove_temp_file(file_name)

    assert content_read == CONTENT

def test_saved_text_file_has_right_content():
    file_name = 'content.txt'
    FileManager.SaveTextFile(file_name, CONTENT)

    content = FileManager.ReadFile(file_name)
    remove_temp_file(file_name)

    assert content == CONTENT
```

Código fonte 32: *test_dependencies.py* - Testes das dependências

5. **Cobertura dos testes:**

   Um dos principais pontos que o grupo deu enfase, foi um grande **coverage** do código, indicando que ele foi amplamente testando, inclindo os casos de borda. Além disso, isso garante uma maior corretude dos testes, bem como um dominio dos integrantes do grupo do código feito.

   Como é possível ver, os arquivos principais apresentam taxa de cobertura bem alta. Além disso, o código todo apresenta uma grande taxa de compressão.

```
---------- coverage: platform linux, python 3.10.12-final-0 ----------
Name                                      Stmts   Miss  Cover   Missing
-----------------------------------------------------------------------
src/BinaryConversor/BinaryConversor.py       13      0   100%
src/BinaryConversor/__init__.py              0      0   100%
src/FileManager/FileManager.py              36     20    44%   5-8, 17-23, 30-36, 40-44
src/FileManager/__init__.py                  0      0   100%
src/LZW/Compressor.py                       85      9    89%   67-68, 74-82
src/LZW/Decompressor.py                     62     10    84%   69-70, 76-86
src/LZW/__init__.py                          0      0   100%
src/Trie/Trie.py                           180     16    91%   40-43, 114-127, 181, 236
src/Trie/__init__.py                         0      0   100%
-----------------------------------------------------------------------
TOTAL                                      376     55    85%
Coverage HTML written to dir htmlcov
```

Figura 4: Coverage do código

6 Análise Experimental

Essa seção irá detalhar os diversos tipos de análise feitas em todo o código, com enfâse nas operações de Compressão e Descompressão. Para isso serão usadas as bibliotecas:

- `cProfile` - Ferramenta integrada ao *Python* para análise de desempenho, que mede o tempo de execução de cada função em um programa. Ele gera relatórios detalhados que ajudam a identificar gargalos de performance. Ideal para otimização de código e diagnóstico de problemas.
- `memray` - Biblioteca poderosa para rastreamento de uso de memória em *Python*. Ele captura dados detalhados sobre alocação e liberação de memória, ajudando a identificar vazamentos e otimizar consumo. Oferece relatórios claros e compatibilidade com visualizações interativas.

6.1 Análise de desempenho

A análise será feita com base no tempo das funções, bem como as taxas de compressão e descompressão. Além disso, os métodos serão análisados com base no seu tempo de execução, número de chamadas e tempo total.

Todos os arquivos dos tópicos a seguir foram gerados pelo `Lorem Ipsum`[2]. Os resultados foram gerados pelo `cProfile`, e foram convertidos em tabelas pelos inte-

[2]https://www.lipsum.com/

grantes do grupo. Além disso todas as analises de tempo serão feita com base em segundos (s).

6.1.1 Explicação de termos:

1. Chamadas: Refere-se ao número de vezes que uma função foi invocada em um programada, ou o número de vezes que o programa executa uma função.

2. Tempo total ou individual: Refere-se ao tempo gasto somando todas as vezes que o programa entrou na função, individualmente, ou seja, é analisado apenas a função e seu escopo, sem contabilizar quando o programa entra em outras funções, mesmo que essas sejam chamadas dentro da função atual analisada.

3. Tempo acumulado: Refere-se ao tempo total que o programa passou dentro de uma função e de todas as funções chamadas por ela.

6.1.2 Compressão fixa

- Arquivo Texto Pequeno - 50kb

Function	Calls	Total Time	Accumulated Time
`Search`	63243	0.453s	0.467s
`ConvertIntegerToBinaryString`	211972	0.092s	0.133s
`Compress`	1	0.086s	0.918s
`Insert`	4095	0.077s	0.090s

Tempo total de execução do programa: 0.950147 segundos **Tamanho Antes:** 50kb **Tamanho Depois:** 20kb

- Arquivo Texto Médio - 562kb

Function	Calls	Total Time	Accumulated Time
`Search`	700329	5.736s	5.896s
`ConvertIntegerToBinaryString`	2464867	1.432s	1.938s
`Compress`	1	0.991s	10.841s
`<listcomp>`	573104	0.715s	2.013s
`ConvertPrefixToBinaryString`	573104	0.333s	2.436s
`builtins.bin`	2464867	0.321s	0.321s

Tempo total de execução do programa: 10.966 segundos **Tamanho Antes:** 562kb **Tamanho Depois:** 185kb

- Arquivo Texto Grande - 5mb

Function	Calls	Total Time	Accumulated Time
`Search`	7,196,219	52.934	54.651
`ConvertIntegerToBinaryString`	20,025,689	12.731	17.523
`Compress`	1	10.508	100.206
`<listcomp>`	5,184,631	5.863	16.552
`ConvertPrefixToBinaryString`	5,184,631	3.159	20.496
`builtins.bin`	20,025,689	3.006	3.006

Tempo total de execução do programa: 101.471 seconds **Tamanho Antes:** 5mb **Tamanho Depois:** 1,17kb

- Arquivo Texto Muito Grande - 40mb

Function	Calls	Total Time	Accumulated Time
`Search`	57,558,783	471.259	486.943
`Compress`	1	93.388	867.494
`ConvertIntegerToBinaryString`	160,343,349	91.929	133.808
`<listcomp>`	41,475,263	53.931	151.042
`ConvertPrefixToBinaryString`	41,475,263	28.809	187.141
`builtins.bin`	160,343,349	26.280	26.280

Tempo total de execução do programa: 882.393 seconds **Tamanho Antes:** 40mb **Tamanho Depois:** 20,8kb

Como é possível ver, o método `Search`, naturalmente, é o método que mais consome tempo individual do processo de compressão, visto que a construção do dicionário é feita por meio de verificações (buscas), seguida pela conversão de inteiro pra binário `ConvertIntegerToBinaryString`, é a segunda que mais consome tempo. A compressão é a que mais gasta tempo acumulado, o que é bem plausível, visto que o processo todo da compressão de um arquivo passa pelo método de `Compress`, o que em todos os casos é a função que representa quase todo o processo da compactção. Além disso outros métodos próprios do *Python* se mostram expressivos, como `<listcomp>` e `builtins.bin`. A razão disso é que o a linguagem tem suas próprias formas de lidar com certos tipos de dados, o que se mostra muito eficiente, por razões práticas. Outro ponto a se notar é que quanto maior for o tamanho do arquivo, maior é tempo gasto pela busca, pela conversão de inteiro pra binário, e o tempo acumulado de compressão. Posteriormente, será mostrado gráficos que elucidam esse aspecto.

6.1.3 Compressão Dinâmica

Neste tópico, iremos analisar a compressão de forma dinâmica, com máximo de bits usados na criação da tabela do LZW igual a 16. O máximo, por padrão é **12 bits**, mas é possível redefini-lo. Além disso, as análises irão se restringir apenas para arquivos grandes e arquivos muito grandes, uma vez que as análise e tabelas se comportam de forma semelhante à compressão fixa.

- Arquivo Texto Grande - 5mb

Function	**Calls**	**Total Time**	**Accumulated Time**
`Search`	6,415,781	98.354	100.424
`Compress`	1	17.993	184.720
`<genexpr>`	33,582,682	17.284	36.868
`ConvertIntegerToBinaryString`	34,198,130	13.390	19.835
`method 'join' of 'str' objects`	15,553,384	13.029	49.898
`builtins.bin`	33,582,425	6.015	6.015
`builtins.ord`	33,582,425	4.203	4.203
`ConvertPrefixToBinaryString`	5,184,631	3.639	47.914
`__getitem__`	6,415,781	2.962	103.386
`Insert`	65,535	2.600	2.976

Tempo total de execução do programa: 185.622 seconds **Tamanho Antes:** 5mb **Tamanho Depois:** 2.6kb

- Arquivo Texto Muito Grande - 40mb

Function	**Calls**	**Total Time**	**Accumulated Time**
`Search`	51,173,809	657.451	671.362
`Compress`	1	121.226	1217.124
`<genexpr>`	271,067,398	116.582	248.270
`ConvertIntegerToBinaryString`	275,916,544	89.158	132.667
`method 'join' of 'str' objects`	124,425,280	85.196	333.467
`builtins.bin`	271,067,141	40.712	40.712
`builtins.ord`	271,067,141	28.618	28.618
`ConvertPrefixToBinaryString`	41,475,263	24.340	322.909
`__getitem__`	51,173,809	18.723	690.086
`GetDetailedReturn`	51,239,344	7.593	7.593

Tempo total de execução do programa: 1222.999 seconds **Tamanho Antes:** 40mb **Tamanho Depois:** 9,4mb

É possível observar, em principio, que a Busca e a Compressão são as mais demoradas, o que não representa nenhuma novidade com relação a compresssão fixa. Porém, difererente de lá, as funções nativas do Python aparecem mais, como `genexpr` e o método `join`. A razão disso são as otimizações feitas pelo grupo, sendo, o `genexpr` um iterador preguiçoso, que não guarda todos os valores em memória ao mesmo tempo, e o `join` é a manipulação de strings. O grupo optou por usar o `join` pois a concatenação de strings em Python é extremamente custosa. Em vez disso, usamos manipulações de listas, que se mostra mais eficiente.

Além disso, o tempo de execução do programa aumenta substancialmente. Esse fato decorre que a tabela a ser criada possui mais entradas a serem preenchidas, o que faz com que o algoritmo continua fazendo buscas e inserções, diferentemente do que ocorre na compressão fixa, uma vez que na fixa, quando a tabela enche, por decisão de projeto, o grupo não realiza mais inserções, e sim apenas buscas para substituições. Logo, a busca se limita a um numero menor de entradas na tabela, e a inserção é limitada mais ainda, pois não é mais feita. Essa margem de limite se torna diferente na compressão dinâmica, com limites de bits. É importante notar que embora a compressão seja dinâmica, o aumento no tempo de execução também aconteceria na compressão fixa, caso o máximo de bits seja maior que 12.

6.1.4 Imagens

O tratamento para de imagens se mostra levemente diferente, pois imagens em Bitmap possuem formato que não é diretamente mapeado para ASCII. Logo, primeiro é feita a conversão para formato ASCII. Após isso a compressão ocorre normalmente.

- Imagem com 148 kb

Function	**Calls**	**Total Time**	**Accumulated Time**
`Search`	1,402,135	27.105	27.498
`<genexpr>`	24,616,175	10.679	23.366
`ConvertIntegerToBinaryString`	24,711,635	7.759	11.104
`method 'join' of 'str' objects`	3,632,375	5.323	28.972
`Compress`	1	3.480	62.235
`builtins.bin`	24,767,256	3.330	3.330
`builtins.ord`	24,615,918	2.363	2.363
`ConvertPrefixToBinaryString`	1,210,961	0.696	27.672
`__getitem__`	1,402,135	0.559	28.057
`Insert`	4,095	0.235	0.255

Tempo total de execução do programa: 62.706 seconds

- Imagem com 7mb

Function	Calls	Total Time	Cumulative Time
`Search`	70,803,211	779.314	798.293
`<genexpr>`	398,240,884	168.707	353.335
`Compress`	1	159.953	1547.491
`ConvertIntegerToBinaryString`	404,150,482	125.513	184.834
`method 'join' of 'str' objects`	176,950,775	115.710	476.494
`builtins.bin`	405,613,565	56.065	56.065
`builtins.ord`	398,240,627	39.046	39.046
`ConvertPrefixToBinaryString`	58,983,761	34.115	456.634
`__getitem__`	70,803,211	27.172	825.465

Total program execution time: 1561.626 seconds

Como é possível notar, as imagens demoram mais para serem compridas. O motivo disso é que os padrões dela são bem diferentes e variam mais que os de textos (que foram gerados pelo `Lorem Ipsum`. De modo geral, a busca é a que mais consome tempo,enquanto a função interna do Python `<genexpr>`, o iterador otimizado, se mostra em segundo lugar. A razão disso deve ser a iteração em grandes strings binárias da imagem, que devem ocorrer com frequência.

Por fim, as **taxas de compressão permeiam em torno de compresão de 1/2 a 3/4 do tamanho original do arquivo**. Isso se mostra bem eficiente para o algoritmo aplicado, tendo uma boa taxa de compressão. Tal ponto será elucidado melhor em seções posteriores.

6.1.5 Descompressão

- Arquivo de entrada de 562kb

Function	Calls	Total Time	Cumulative Time
`Insert`	126,739	2.116	2.975
`Search`	126,484	0.927	0.955
`__init__`	253,477	0.591	0.614
`Decompress`	1	0.352	4.690
`built-in method builtins.max`	126,747	0.182	0.182
`ConvertIntegerToBinaryString`	126,739	0.124	0.149
`<genexpr>`	189,730	0.087	0.128
`__getNextCode`	126,484	0.066	0.109

__setitem__	126,739	0.055	3.030
built-in method builtins.bin	316,468	0.047	0.047

Total program execution time: 4.886 seconds

- Arquivo de entrada 5MB

Function	Calls	Total Time	Cumulative Time
Insert	615,959	18.679	23.466
Search	615,704	10.155	10.381
init	1,231,917	2.999	3.180
Decompress	1	2.524	38.839
built-in method builtins.max	615,967	1.120	1.120
<genexpr>	1,231,413	1.019	1.444
Decompressor.__getNextCode	615,704	0.551	0.701
ConvertIntegerToBinaryString	615,959	0.639	23.898
setitem	615,959	0.454	23.898
built-in method builtins.bin	1,847,371	0.412	0.412

Total program execution time: 40.700 seconds

- Arquivo de entrada Imagem de 7mb

Function	Calls	Total Time	Cumulative Time
Insert	5,910,109	202.312	247.590
Search	5,909,854	70.136	72.155
init	11,820,217	33.033	34.611
Decompress	1	22.543	364.619
<genexpr>	8,864,786	7.473	10.633
built-in method builtins.max	5,910,117	6.108	6.108
Decompressor.__getNextCode	5,909,854	4.773	6.110
SaveTextFile	1	4.481	6.931
ConvertIntegerToBinaryString	5,910,109	4.359	6.126
setitem	5,910,109	4.015	251.606

Total program execution time: 384.751 seconds

Como é possível ver, de modo geral, o método `Insert` é o que mais gasta tempo. O motivo disso se dá pelo fato de que todos os códigos devem ser inseridos na

tabela, afim de reconstruir o código, enquanto na compressão, isso não ocorre sempre. O método `Search`, como esperado, consome muito tempo também, bem como o `__init__`, que se dá pelo fato de serem criados mais nós (em essencia) na árvore, e outros *inits* de arquivos e outras classes.

Outro ponto notório é o menor tempo de execução. Tal razão se da pelo fato de ter que gerar bem menos códigos o binário do arquivo(que naturalmente é menor), bem como o alto número de inserções, que experimentalmente, se mostra menos custosa que a busca. Somando isso, a descompressão é feita em muito menos tempo.

6.2 Análise de Memória

A análise de memória serviu para identificar o consumo total de memória alocada durante o processo de compressão, além de observar padrões e identificar os principais responsáveis pelo uso de memória no sistema.

Todos os arquivos dos tópicos a seguir foram gerados pelo `Lorem Ipsum`. Os resultados foram gerados pelo `memray`, e foram convertidos em tabelas pelos integrantes do grupo.

- Arquivo Input de 562kb - Compressão fixa

  Top 5 largest allocating locations (by size):

  - ConvertIntegerToBinaryString: 9.001MB
  - __convertCodesToBinaryString:8.677MB
  - Compress: 8.677MB
  - ReadFile: 2.191MB
  - SaveBinaryFile: 1.633MB

Location	Total Memory	Total Memory %	Own Memory
`ExecuteCompressOperation`	14.932MB	100.00%	0.000B
`main`	14.932MB	100.00%	0.000B
`<module>`	14.932MB	100.00%	0.000B
`Compress`	14.385MB	96.33%	990.000KB
`__convertCodesToBinaryString`	10.416MB	69.75%	2.414MB
`ConvertIntegerToBinaryString`	9.001MB	60.28%	9.001MB
`__setitem__`	1.003MB	6.72%	592.000B
`Insert`	1.002MB	6.71%	2.256KB
`__insertNewCode`	1.002MB	6.71%	0.000B
`__init__`	1.000MB	6.70%	1.000MB

<listcomp>	1.000MB	6.70%	1.000MB
ConvertPrefixToBinaryString	1.000MB	6.70%	0.000B
ReadFile	559.469KB	3.66%	559.469KB
__init__	592.000B	0.00%	0.000B

- Arquivo Input de 5mb - Compressão Dinâmica

  Top 5 largest allocating locations (by size):

  - __convertCodesToBinaryString: 130.647MB

  - Compress: 30.647MB

  - ConvertIntegerToBinaryString: 111.021MB

  - SaveBinaryFile:24.617MB

  - __convertCodesToBinaryString: 20.398MB

Location	**Total Memory**	**Total Memory %**	**Own Memory**
Compress	162.452MB	97.05%	14.520MB
ExecuteCompressOperation	162.452MB	97.05%	0.000B
main	162.452MB	97.05%	0.000B
<module>	162.452MB	97.05%	0.000B
__convertCodesToBinaryString	145.930MB	87.18%	34.918MB
ConvertIntegerToBinaryString	111.012MB	66.32%	111.012MB
Insert	1.002MB	0.60%	1.002MB
__setitem__	1.002MB	0.60%	0.000B
__insertNewCode	1.002MB	0.60%	0.000B
ConvertPrefixToBinaryString	1.000MB	0.60%	1.000MB

Tabela 13: Memory Usage Analysis

- Descompressão Imagem 7mb

  Top 5 largest allocating locations (by size):

  - __init__: 1.538GB

  - ReadBinaryFile: 747.340MB

  - ¡genexpr¿: 549.091MB

  - ConvertIntegerToBinaryString:inaryConversor.py: 489.000MB

  - Decompress:py: 424.325MB

Location	Total Memory	Total Memory %	Own Memory
ExecuteDecompressOperation	3.000GB	100.00%	536.000B
main	3.000GB	100.00%	0.000B
<module>	3.000GB	100.00%	0.000B
Decompress	2.867GB	95.56%	412.403MB
Insert	1.924GB	64.14%	395.188MB
setitem	1.924GB	64.14%	0.000B
init	1.538GB	51.27%	1.538GB
ConvertIntegerToBinaryString	489.000MB	15.92%	489.000MB
ReadBinaryFile	68.633MB	2.23%	67.633MB
ExtractCodeLengthAndContent	67.633MB	2.20%	67.633MB
_getNextCode	64.000MB	2.08%	64.000MB
<genexpr>	1.000MB	0.03%	1.000MB

Como é possível notar, as maiores alocaçoes de espaço são feitas para Conversão para String binárias, e os *inits*. A razão disso é o proprio fato de 1) leitura e gerenciamento de arquvios, para a geração das strings binárias e 2) a inicilização de várias classes e estruturas de dados que consomem memória.

Além disso, em relação à memória do programa, tanto a compressão fixa/dinãmica, como a descompressão gastam a maioria da memória do programa, mas não individualmente e sim a memória gasta enquanto o programa está na função. Indivualmente, a conversão de string para binário se mostra a mais custosa, em média, para espaço, enquanto a compressão não parece impactar tanto na memória, o que é previsível, já que a própria operação de compressão gera arquivos com menos memória. Já a descompressão se torna quase que igualada, em relação à gasto de espaço com a conversão de strings, o que é natural, pela própria definição da descompressão.

6.3 Estatísticas coletadas durante a execução

Nesse sessão, serão exibidos os dados coletados durante a execução do algoritmo de compressão e descompressão para alguns tipos de arquivos, como textos e imagens em formato bitmap. Esses dados serão compostos por informações internar do funcionamento dessa ferramenta, como o tempo transcorrido nas operações, uso de memória para o dicionário (representado pela *trie*) e taxas de compressão e descompressão. Para a coleta e análise dessas informações foram usadas diversas bibliotecas, como o *pandas* para agregação dos dados e o *seaborn* para o plot das imagens.

6.3.1 Imagem em formato bitmap – 148 KB

As Figuras 5 e 6, evidenciam a eficiência e a simetria entre os processos de compressão e descompressão para o arquivo BMP de 148 KB. Durante a compressão, a taxa de compressão cresce até estabilizar, enquanto o progresso avança linearmente, refletindo o processamento contínuo do algoritmo. O tamanho do dicionário aumenta gradualmente, indicando o aprendizado de novos padrões. Já na descompressão, o progresso mantém um padrão linear, enquanto o tamanho do dicionário estabiliza rapidamente.

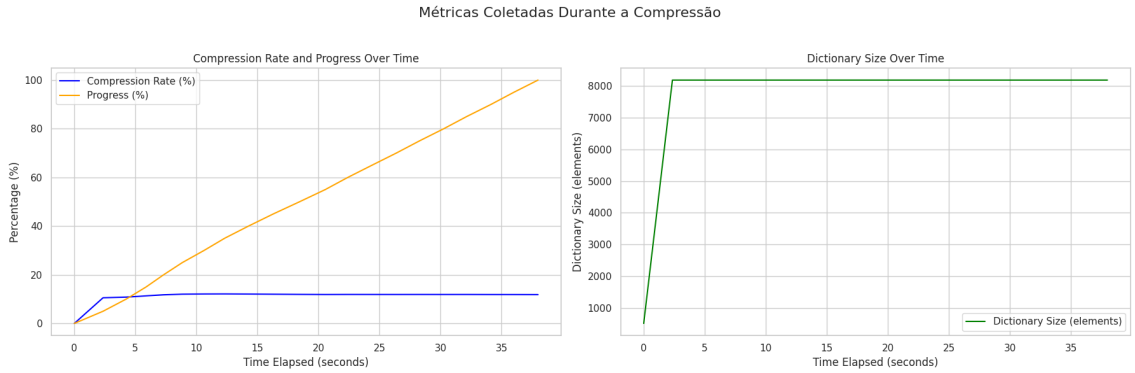

Figura 5: Estatísticas de compressão para um arquivo BMP de 148KB.

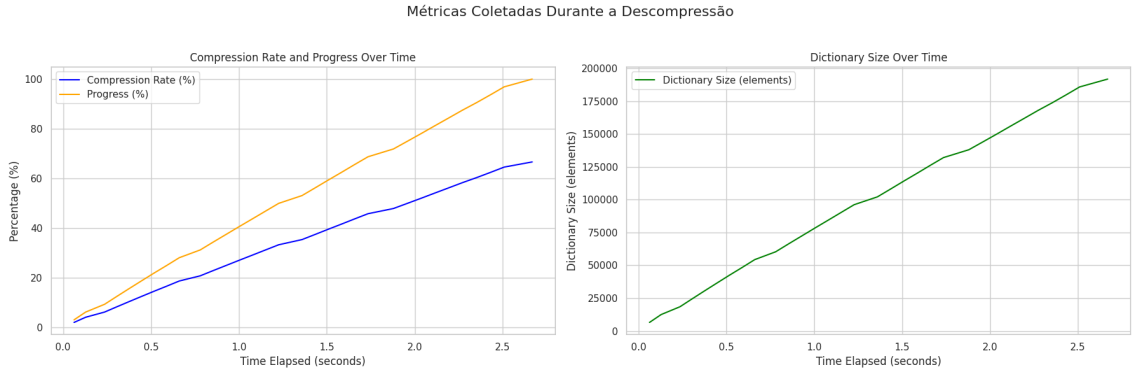

Figura 6: Estatísticas de descompressão para um arquivo BMP de 148KB.

6.3.2 Arquivo de texto com 40MB

As Figuras 7 e 8 ilustram o comportamento do algoritmo ao comprimir e descomprimir um arquivo de texto de **40MB**, reduzindo-o de *40506KB* para *20874KB*. Durante a compressão, nota-se que a taxa de compressão cresce rapidamente no início, estabilizando-se conforme o dicionário alcança seu limite de crescimento, imposto pela quantidade de bits disponíveis para representar as palavras. Esse limite restringe a identificação de novos padrões, o que explica a estabilização observada. O progresso da compressão avança linearmente, refletindo o processamento consistente do arquivo. Já na descompressão, o progresso segue igualmente linear, enquanto o dicionário rapidamente atinge seu tamanho máximo.

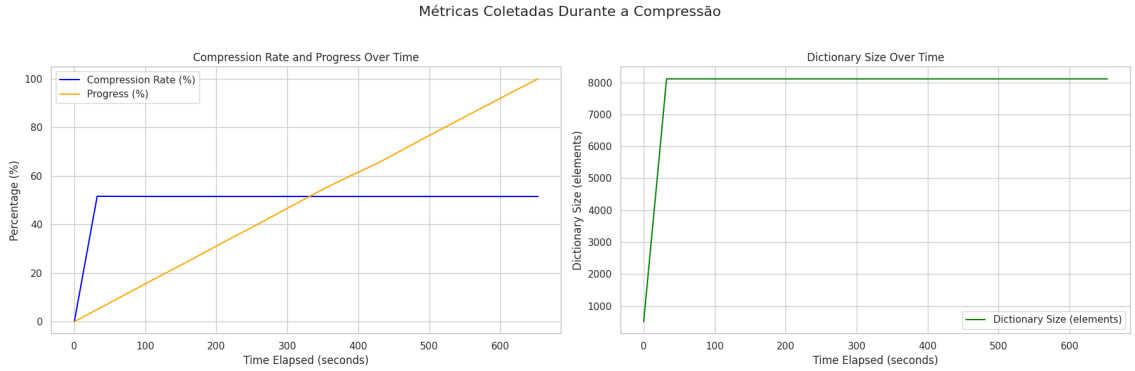

Figura 7: Estatísticas de compressão para um arquivo texto de 40MB

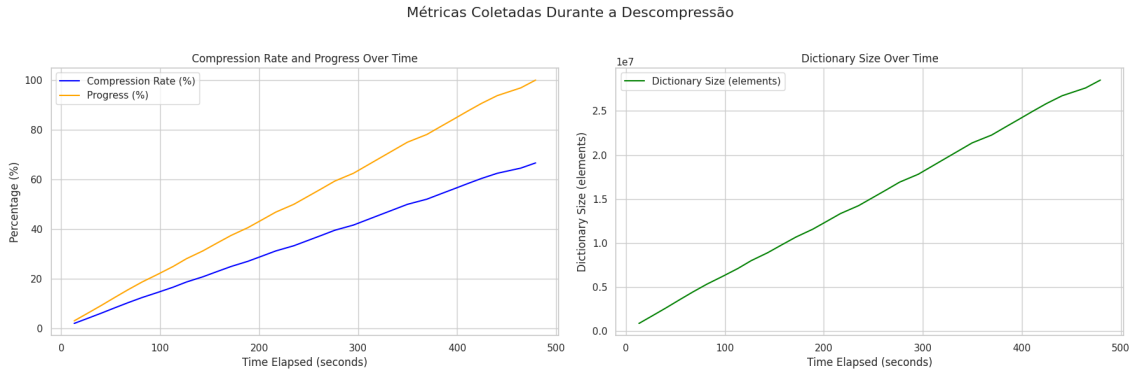

Figura 8: Estatísticas de descompressão para um arquivo texto de 40MB

6.3.3 Comparação entre compressão fixa e dinâmica

As Figuras 9 e 10 apresentam os resultados da compressão de um arquivo de *6125KB* utilizando **códigos fixos** e **dinâmicos**, reduzindo-o para *2026KB* e *1507KB*, respectivamente. Observa-se que a compressão com códigos dinâmicos atinge uma taxa de compressão mais eficiente, devido à capacidade do algoritmo de adaptar os padrões do dicionário às características específicas do arquivo. Por outro lado, a compressão com códigos fixos demonstra uma estabilização mais rápida na taxa de compressão, limitada pelo tamanho máximo do dicionário. O progresso, em ambos os casos, avança linearmente, mas o tamanho do dicionário na abordagem dinâmica cresce de maneira mais eficiente, resultando em uma maior redução no tamanho final do arquivo.

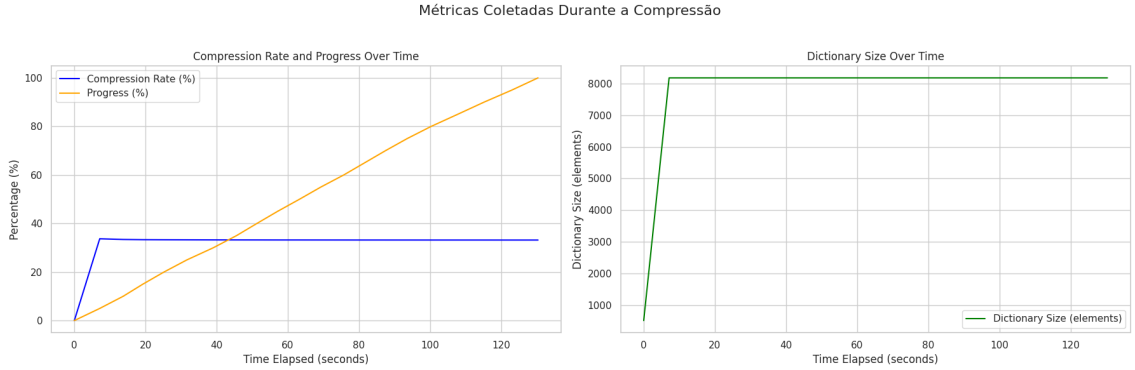

Figura 9: Métricas de compressão usando códigos fixos

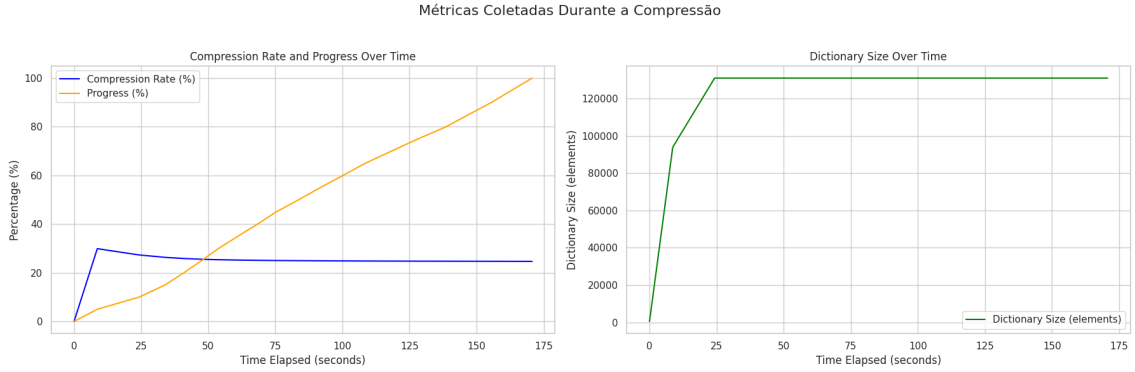

Figura 10: Métricas de compressão usando códigos dinâmicos

7 Conclusões

Em suma, o código feito está de acordo com a especificações. Para cumprir tais especificações, foi de fundamental importância a definição da estruturas de dados a serem utilizadas, bem como a plena compreensão do método LZW. Além disso, como o código lida com um grande volume de dados, foi necessários refatorações e otimizações de códigos, para torná-lo mais eficiente. Para isso, as ferramentas de análise temporal `cProfile` e espacial `memray` foram de fundamental importância. Sendo assim, por meio da implementação e análise de resultados de testes, fica comprovado a eficiência do método utilizado, bem como é elucidado a importância dos algoritmos vistos em sala para a resolução de outros problemas diferentes dos apresentados em aula.

Referências

[1] UNIVERSIDADE ESTADUAL DE CAMPINAS. *Algoritmo LZW*. Disponível em: `https://www.decom.fee.unicamp.br/dspcom/EE088/Algoritmo_LZW.doc`. Acesso em: 8 nov. 2024.

[2] GEEKSFORGEEKS. *Compressed Tries*. Disponível em: `https://www.geeksforgeeks.org/compressed-tries/`. Acesso em: 6 nov. 2024.

[3] WIKIPÉDIA. *Árvore Patricia*. Disponível em: `https://pt.wikipedia.org/wiki/%C3%81rvore_Patricia`. Acesso em: 6 nov. 2024.

[4] LUCET, Yves. *Compressed Trie*. Disponível em: `https://cmps-people.ok.ubc.ca/ylucet/DS/CompressedTrie.html`. Acesso em: 6 nov. 2024.

[5] CSC. *cProfile Documentation*. Disponível em: `https://docs.csc.fi/computing/cProfile/`. Acesso em: 15 nov. 2024.

[6] BLOOMBERG. *Memray Documentation*. Disponível em: `https://bloomberg.github.io/memray/getting_started.html`. Acesso em: 15 nov. 2024.

[7] WIKIPÉDIA. *Análise amortizada*. Disponível em: `https://pt.wikipedia.org/wiki/An%C3%A1lise_amortizada`. Acesso em: 19 nov. 2024.