Trabalho Prático 1
Algoritmos 2

Daniel Ferreira Abadi[1] **- 2018088062**

[1] Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG) – Belo Horizonte, MG – Brasil

1. Introdução

O problema abordado consiste em implementar um algoritmo para resolver o problema de comprimir arquivos de texto através do método LZ78. E para esse algoritmo é comum utilizar árvores de prefixo na sua implementação. Nesse trabalho em específico foi nos dada a tarefa utilizando-se a Trie Compacta. E, também, realizar a descompressão dos arquivos comprimidos, podendo utilizar uma estrutura que seja ideal para tal ação.

2. Implementação da Compressão

Como nos foi passado, teríamos de utilizar uma Trie Compacta para a lógica do LZ78. Para isso foram criadas duas classes, chamadas TrieNode e Trie. E neste trabalho foi utilizada a linguagem Python, versão 3.6.5.

2.1. Classe TrieNode

A classe TrieNode apenas guarda elementos básicos de um nó, sendo uma ”string”, para armazenar o prefixo, um vetor, para armazenar os índices de cada caractere adicionado ao nó, um dicionário que guarda a relação caractere e filho, e o nó pai.

Esses índices são relativos ao algoritmo LZ78, começando com 0 e sendo incrementado a cada caractere adicionado na trie.

2.2. Classe Trie

A classe Trie é a implementação do método LZ78, ela possuí um nó próprio, sendo a raiz, e a função ”insere”, que faz todo o trabalho de inserção na trie, como o algoritmo LZ78 define.

A função de inserção recebe 3 parâmetros, o texto a ser inserido, a posição atual do texto e o arquivo de destino. A função em si, funciona como se fosse um autômato finito, se ela insere um novo caractere na trie a função ”reinicia”, voltando para o nó inicial.

O método funciona basicamente desta forma: verificamos se o caractere é igual ao primeiro caractere do rótulo do nó, e caso seja igual, verificamos os próximos. Se todo o rótulo for igual a ”substring”do texto e não tiver filhos, então o próximo índice do texto é adicionado ao rótulo daquele nó. Mas se o nó tiver filhos, o próximo índice do texto é buscado entre os filhos, caso haja um filho com aquela chave, vamos para o filho e voltamos ao início da função. Se não tiver nenhum filho atrelado àquele índice, então um novo nó é criado. Caso apenas uma parte do rótulo seja igual, então ocorre uma divisão daquele nó. O nó original perde seus filhos e fica apenas com o prefixo que ”casou”com o texto, mas recebe como filho os novos dois nós, sendo o nó contendo o sufixo restante do

nó original, que recebe todos os filhos pré existentes, e o nó que recebe o caractere vindo do texto.

Caso o primeiro caractere do rótulo seja diferente do índice do texto, então é verificado se há algum filho com aquele índice, se existir então vamos para o nó filho e voltamos para o início da função. Caso não exista, então criamos um novo nó filho e o atrelamos àquele caractere, reiniciando a função.

Uma observação importante é que o método realizado não comprimia realmente os arquivos, normalmente aumentavam-se os tamanhos dos mesmos. Com arquivos menores esse aumento chegava a ser o dobro do tamanho original, e com arquivos maiores chegava a um aumento de 20 porcento. Tendo isso em vista, decidi que o melhor seria passar os dados para o arquivo em binário, para realmente comprimir os arquivos.

Há vários pequenos detalhes na implementação da função, porém considero que o essencial está logo acima, não precisando dos detalhes pequenos que ficaram de fora. Sendo esses tratamentos em relação ao texto acabar no meio da árvore.

3. Implementação da Descompressão

Para a descompressão temos, inicialmente, um dicionário, que na primeira posição recebe a ”string”vazia. A partir daí recebemos os valores do arquivo de leitura, sendo uma dupla de índice e caractere, e o adicionamos no dicionário, na ordem que nos foi dada no arquivo. Devido a como a compactação foi feita, temos que o casamento entre os índices dos pares e os índices do dicionário casam perfeitamente.

A cada dupla lida, é chamada uma função que recebe o dicionário, o índice e o arquivo de escrita. O método guarda o caractere associado àquele valor no início de uma ”string”, inicialmente vazia. Logo em seguida pega o valor associado ao caractere e continua o processo até chegar na posição 0 do dicionário, que retorna vazio. Por exemplo, se um par tem o índice como 1, podemos ir na posição 1 do dicionário e pegar os dados de lá para completar a ”string”, e assim por diante até chegar a posição 0 do dicionário.

Uma informação importante é que os dados recebidos pelo arquivo estão em bytes, portanto todos são traduzidos para inteiro e caractere, e só então escritos no arquivo.

4. Análise experimental

Foram utilizados 10 livros do site ”https://www.gutenberg.org”, sendo todos em português de diferentes tamanhos, como especificado. Há o detalhe adicional de que ambos os arquivos originais e descompactados devem estar no formato ”LF”em relação ao tipo do caractere usado para representar a quebra de linha.

O arquivo 1, original, tem 58 KB e após a compressão foi para 69 KB, uma taxa de aproximadamente 0.84 de compressão.

O arquivo 2, original, tem 83 KB e após a compressão foi para 93 KB, uma taxa de aproximadamente 0.89 de compressão.

O arquivo 3, original, tem 210 KB e após a compressão foi para 211 KB, uma taxa de aproximadamente 0.99 de compressão.

O arquivo 4, original, tem 337 KB e após a compressão foi para 323 KB, uma taxa de aproximadamente 1.05 de compressão.

O arquivo 5, original, tem 339 KB e após a compressão foi para 316 KB, uma taxa de aproximadamente 1.07 de compressão.

O arquivo 6, original, tem 386 KB e após a compressão foi para 364 KB, uma taxa de aproximadamente 1.06 de compressão.

O arquivo 7, original, tem 401 KB e após a compressão foi para 370 KB, uma taxa de aproximadamente 1.08 de compressão.

O arquivo 8, original, tem 434 KB e após a compressão foi para 406 KB, uma taxa de aproximadamente 1.07 de compressão.

O arquivo 9, original, tem 569 KB e após a compressão foi para 513 KB, uma taxa de aproximadamente 1.11 de compressão.

O arquivo 10, original, tem 767 KB e após a compressão foi para 604 KB, uma taxa de aproximadamente 1.27 de compressão.

Tendo esses dados em mãos, é notável que o algoritmo LZ78 não funciona bem para arquivos pequenos, podendo expandir seu tamanho, aumentando sua eficiência em arquivos médios e grandes. Mas dependendo muito da estrutura textual do conteúdo do mesmo.

5. Conclusão

Foi implementado o algoritmo de compressão de arquivos LZ78 utilizando-se de uma trie compacta e uma método para a descompressão. Para a compressão foram ciadas duas classes, TrieNode, representando o nó, e Trie, a estrutura da árvore com a função de inserção. Para a descompressão foi utilizado da estrutura da compressão que resultou em uma relação perfeita com um dicionário Python.

As dificuldades iniciais foram entender o método de inserção da trie, que são complexos em alguns casos como a separação de sufixos e entender como o algoritmo LZ78 funciona. Um contratempo que me levou algumas horas de trabalho foi em relação ao texto poder acabar estando no meio da trie, o que poderia gerar diversos erros e situações que fazem o algoritmo dar uma resposta errada.

Para trabalhos posteriores fica a experiência do uso da linguagem Python, visto que nunca tinha a utilizado para projetos como este.