Introdução

O objetivo deste trabalho é desenvolver um algoritmo para realizar a compressão e descompressão de arquivos de texto utilizando o método LZ78. Esse método se baseia na construção de um dicionário para armazenar strings que se repetem no texto a fim de substituí-las por códigos, reduzindo assim o número de bytes gravados na saída.

Como o algoritmo LZ78 executa muitas buscas e inserções no dicionário, foi proposta a utilização de árvores de prefixos na sua implementação. Portanto, no lugar de um dicionário, implementamos uma Trie tradicional (árvore de prefixos), onde cada nó da árvore armazena um **caractere** do texto, um **código** e um **id**, além de uma lista contendo seus nós filhos.

É importante observar que um nó **B**, que armazena um caractere **b**, é filho do nó **A**, que por sua vez armazena o caractere **a**, se **ab** é uma sequência de caracteres lidos do texto. Dessa forma, cada nó armazena apenas um caractere, mas o caminho que parte da raiz até este nó representa uma sequência de caracteres que foi inserida na árvore. Assim, o **código** de um nó refere-se ao código da sequência de caracteres inserida, e não apenas ao caractere que está neste nó. Já o **id** de um nó corresponde à ordem em que a sequência de caracteres foi inserida na árvore, de modo que a primeira sequência inserida tenha id 1, a segunda id 2, e assim por diante. Este id será necessário para definirmos o código de uma nova sequência inserida na árvore e para determinarmos a ordem que os pares (código; caractere) devem ser impressos no arquivo de saída. Discutiremos isso mais adiante, quando analisarmos o algoritmo para a compressão de um arquivo.

Compressão

Para realizar a compressão do arquivo, primeiramente precisamos ler seu conteúdo, nos certificando que seja usada a codificação UTF-8, e armazená-lo em uma string. Em seguida precisamos construir a árvore de prefixos que armazenará os caracteres e códigos que serão impressos no arquivo de saída.

Para construir a árvore, primeiramente, precisamos inicializar a raiz da Trie com a string vazia e **id** e **código** iguais a 0. Assim, ao ler um caractere **c** nunca antes lido pelo programa, ele será associado ao código 0, como ainda veremos. Então, processaremos o texto da esquerda para a direita, caractere a caractere. Ao ler um novo caractere, verificamos se ele já ocorreu na Trie. Em caso positivo, concatenamos o próximo caractere do texto à cadeia atual e verificamos se a nova sequência se encontra na árvore. Enquanto as sequências já estiverem presentes na árvore, o algoritmo continua lendo e concatenando. Quando lermos um caractere **c** que concatenado à sequência não está presente na árvore, precisamos gerar o código da nova sequência terminada em **c** e inseri-la na árvore. Mas note que com exceção do caractere **c**, todos os demais caracteres da sequência já estão armazenados na árvore, afinal este é o primeiro caractere a "quebrar" a sequência. Portanto, na prática, precisamos adicionar apenas o nó

correspondente ao caractere **c** ao final do ramo da sequência na árvore, isto é, no seu último nó.

Dessa forma, o nó que armazenará o caractere **c** deve ter **código** igual ao **id** da sequência em que está sendo adicionado, indicando que o par (código; caractere) representa a cadeia formada pelos prefixo das sequências inseridas anteriormente naquele ramo concatenada ao caractere do nó atual. Como o id representa a ordem de inserção em que as cadeias foram inseridas, se imprimirmos no arquivo de saída os pares (código; caractere) na ordem em que foram descobertos no texto, ou seja, ordenadas pelos **id'**s, seremos capazes de reconstruir a árvore durante a descompressão do arquivo, pois saberemos que o caractere associado ao código **n**, por exemplo, deve ser armazenado em um nó filho do nó que contém **id n**, isto é, o nó que foi a **n**-ésima inserção na árvore. Por isso o **id** da raiz, que contém a string vazia, deve ser 0, para que todos seus nós filhos possuam **código** 0 e sejam inseridos na lista de filhos da raiz durante uma futura descompressão.

Dito isto, quando uma sequência não for encontrada na árvore, devemos adicionar seu último caractere ao final de seu ramo na árvore e reiniciar o processando a partir do próximo caractere não lido do texto. O ciclo se repete até que todo o arquivo tenha sido processado.

Precisamos então de uma função que verifique se uma dada sequência se encontra na Trie. Tal função, chamada **find_by_label**, recebe como parâmetros um nó **N** da árvore e uma sequência **S** de caracteres. Caso **N** seja nulo, a sequência não se encontra na árvore, então retornamos nulo. Caso contrário, se **N** for a raíz da Trie, verificamos em qual de seus filhos está armazenado o primeiro caractere da cadeia **S** dada e prosseguimos pela busca de **S** a partir deste filho, pois nenhum caractere foi casado ainda. Porém se **N** não é a raiz, mas o caractere armazenado por ele for igual ao primeiro caractere de **S** verificamos em qual de seus filhos está armazenado o próximo caractere da cadeia, caso ele exista, e seguimos buscando pelos caracteres ainda não casados a partir desse nó filho; porém caso **S** tenha apenas um caractere, a cadeia se encontra na árvore e seu último nó é este, logo o retornamos. Por fim, se **N** não é a raiz mas caractere armazenado por ele difere do primeiro caractere de **S**, **S** não está armazenada na árvore e retornamos nulo.

Agora que temos uma função para verificar se uma sequência está armazenada na Trie, podemos definir uma função **make_tree** que constrói a árvore a partir do texto. Primeiramente inicializamos a raiz da Trie com a string vazia e **id** e **código** iguais a 0. Então processamos o texto caractere a caractere. Ao ler um caractere, buscamos seu nó correspondente na árvore. Caso o nó retornado seja nulo, o caractere não está armazenado na árvore, logo deve ser inserido na raiz. Caso contrário, enquanto a cadeia atual acrescida do próximo caractere do texto estiver armazenada na árvore, concatenar próximo caractere **c** do texto à nova sequência e buscá-la na árvore a partir da raiz: se nó retornado for nulo, a adição do caractere **c** "quebrou" a sequência, logo devemos adicioná-lo aos filhos do último nó não nulo retornado – ele é o último nó correspondente à parte da sequência que já estava armazenada na árvore.Repetimos o ciclo até que todo o texto seja processado.

Contudo, um cuidado extra deve ser tomado ao processarmos a última sequência de caracteres do texto. Caso essa sequência já esteja armazenada na árvore, sairemos do loop sem adicioná-la novamente, mas o deveríamos, para que possamos recuperá-la durante uma futura descompressão do arquivo. Dessa forma, devemos verificar se terminamos de processar o texto sem inserir a última cadeia: em caso positivo, devemos encontrar o nó onde deve ser inserido o último caractere da cadeia e fazê-lo.

Note que, como sempre adicionamos apenas um nó a cada inserção na árvore e sempre sabemos onde o novo nó será inserido, a função **insert** para inserir caracteres na árvore simplesmente cria um novo nó com os parâmetros passados e o acrescenta à lista de filhos do nó passado como parâmetro.

Uma vez construída a árvore, iremos gerar os dados a serem impressos no arquivo de saída. Iremos gerar uma string de bits, contendo, entre outras coisas, a representação binária dos pares (codigos; caractere), que posteriormente será convertida em bytes de acordo com seu tamanho e, finalmente, escrita em no arquivo binário de saída.

Para isto, precisamos primeiramente recuperar os pares (código; caractere) para gerar a saída. Definimos então a função **get_nodes** que percorre toda a árvore adicionando a uma lista chamada **output** um array de três posições contendo o id, o código e o caractere armazenado, respectivamente, de cada nó da árvore. Em seguida, ordenamos esta lista de acordo com o **id** (pois é o primeiro elemento de casa sublista em **output**), para que possamos imprimir os pares (código; caractere) na saída conforme a ordem que foram descobertos durante o processamento do texto.

Feito isto, computamos o número de bits necessários para representar os códigos de cada par da saída. Ele será igual à função teto de $\boldsymbol{log_2 C}$, onde **C** é o número total de códigos gerados. O número de bits precisa ser colocado no início do arquivo para que possamos fazer a leitura correta dos bits durante a descompressão. Como esse valor é diferente para cada arquivo, usaremos sempre 20 bits para representá-lo. Porém, com isso, os zeros à esquerda serão ignorados quando lermos os bits de volta durante a descompressão, logo temos que acrescentar um ‘1’ no início do arquivo para evitar que isto aconteça. Além disso, os caracteres associados a um código podem precisar ser representados utilizando diferentes número de bytes. Por exemplo, caracteres ASCII são representados apenas com um byte, ao passo que caracteres especiais podem precisar de dois ou mais bytes para serem devidamente codificados. Dessa forma, acrescentamos ao início de cada código, dois bits para indicar quantos bytes o caractere associado possui (1, 2, 3 ou 4), aumentando o número necessário de bits para representar um código em duas unidades.

Então, iniciaremos nossa string de bits com um ‘1’, para preservar os 0s à esquerda, e logo em seguida, acrescentaremos a representação binária do número de bits necessários para se representar os códigos de fato mais os dois bits para sinalizar a quantidade de bytes de um caractere.

Agora, vamos então gerar a string de bits. Para cada sublista correspondente a um nó na lista ordenada **output**, vamos codificar o caractere, concatenar à string de bits os bits que representam o número de bytes que o caractere possui, os bits que representam o código, e os bits que representam o caractere, respectivamente.

Após codificar todos os pares (código; caractere), calculamos o número de bytes que serão necessários para representar nossa string de bits, que é igual ao teto da quantidade de bits na string dividida por 8. Por fim convertemos a string de bits para bytes, escrevemos seu conteúdo no arquivo de saída do programa e encerramos a execução do programa.

Descompressão

A descompressão do arquivo segue o mesmo algoritmo. Primeiramente precisamos ler o conteúdo do arquivo, obter sua representação binária convertendo os bytes lidos em bits, e armazená-la em uma string **bits**. Em seguida, precisamos descobrir quantos bits são usados para representar cada código. Para isto, descartamos o primeiro bit da string, pois ele serve apenas para preservar os zeros à esquerda, e lemos os próximos 20 bits, referentes ao tamanho de cada código.

Agora, precisamos construir a árvore de prefixos que armazenará os caracteres que serão impressos no arquivo de saída. Novamente, inicializamos a raiz da árvore com a string vazia associada ao id e código 0. Então, para cada par (código; caractere) presente na string de bits, lemos e convertemos para inteiro os dois bits que indicam quantos bytes o caractere possui, lemos e convertemos para inteiro os bits que representam o código, lemos e convertemos para bytes os bits correspondentes ao primeiro byte do caractere. Se o caractere estiver sendo representado por mais de um byte, lemos os bytes seguintes e concatenamos aos anteriores. Após ler todos os bytes do caractere, o convertemos para string usando a codificação UTF-8 e buscamos em qual nó da árvore o novo nó correspondente a este caractere será inserido. Para isto, definimos a função **find_by_code** que recebe como parâmetros a raiz da árvore e o código associado ao caractere que queremos inserir. Esta função percorre os nós da árvore e retorna aquele que possui **id** igual ao **código** que foi passado como parâmetro. É importante observar que nesta busca, precisamos visitar somente os nós cujo código é menor que o código que foi passado como parâmetro, aquele associado ao novo nó que queremos inserir, pois

Por fim, caso o nó tenha sido encontrado, inserimos o caractere corrente usando a mesma função **insert** utilizada na compressão do arquivo.

Agora precisamos recuperar os caracteres que deverão ser impressos na saída do arquivo. Isto é feito utilizando a função **get_text** que percorre todos os nós da árvore acumuluando o rótulo dos nós por onde passa em uma string chamada **prefix** e adicionando a uma lista chamada **output** um array de três posições contendo o **id**, o **código** e o **prefix** correspondente de cada nó da árvore, de forma que **prefix** armazene a sequência de caracteres presentes no caminho que começa na raiz e termina no nó corrente. Ou seja, na prática, estamos

recuperando todos os ramos da árvore, e consequentemente, todas as cadeias de caracteres que foram inseridas nela, com seus respectivos ids e códigos.

Em seguida, ordenamos a lista output de acordo dos ids dos nós correspondentes. Ao fazer isto, estamos garantindo que as cadeias de caracteres serão impressas na mesma ordem em que foram lidas durante a compressão do arquivo. Então criamos uma string vazia para receber as cadeias de caractere e, para cada sublista em output referente a um nó, concatenamos a sequência armazenada.

Por fim, imprimimos o texto obtido no arquivo de saída e encerramos a execução do programa.

Testes

Foram realizados testes com 13 arquivos textos reais obtidos através do Projeto Gutemberg. A tabela abaixo mostra, para cada arquivo texto, o tamanho do arquivo original, o tamanho do arquivo comprimido e a taxa de compressão observada.

Nome do arquivo	Tamanho original (bytes)	Tamanho comprimido (bytes)	Taxa de compressão (%)
bases_da_ortografia_portuguesa	50.213	36.866	0.2658076593710792
chronicas_de_viagem	142.879	94.418	0.33917510620875
cinco_minutos	106.327	70.697	0.3350983287405833
dracula	865.819	495.507	0.4277014017941394
hamlet_-_drama_em_cinco_actos	214.615	137.436	0.3596160566595997
iracema	202.780	129.863	0.3595867442548575
memorias_postumas_de_bras_cubas	394.823	245.862	0.37728551781431174
othello	152.528	96.994	0.3640905276408266
quincas_borba	482.461	288.856	0.40128632158868804
romeo_and_juliet	163.623	107.330	0.3440408744491912
sonetos	36.963	27.949	0.24386548710873035
the_divine_comedy	640.622	376.624	0.4120963688415321
the_republic	1.219.023	642.787	0.4727031401376348

A taxa de compressão média observada é aproximadamente 0.36%