UNIVERSIDADE FEDERAL DE MINAS GERAIS
DCC – DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

TRABALHO PRÁTICO 1

Disciplina: Algoritmos 2

Turma: TN

Professor: Renato Vimieiro

Aluno: Daniel Souza de Campos

Curso: Ciência da Computação

Fevereiro/2021

Sumário

1 Introdução

A compressão de arquivos é muito importante para permitir um armazenamento mais eficiente de tais recursos. Para tal, diversos algoritmos, e programas que fazem uso desses algoritmos, já existem com o objetivo de contribuir para essa área. Talvez, uma das principais características de um algoritmo de compressão é a sua **taxa de compressão.** Essa taxa representa quanto o algoritmo foi capaz de diminuir o tamanho original do arquivo. Exemplo: Um arquivo “original.txt” com um tamanho de 100KB foi comprimido e o arquivo “comprimido.z78” foi gerado de tamanho 80KB. Nesse caso, a taxa de compressão foi de 100/80 = 1.25. Quanto maior essa taxa, melhor o algoritmo de compressão.

2 Objetivo

O objetivo do trabalho é desenvolver um programa que comprima arquivos de texto. Mais especificamente, deve-se basear o algoritmo seguindo o algoritmo já existente chamado **LZ78**. A ideia do algoritmo LZ78 é substituir

padrões no texto do arquivo original por códigos e, desse modo, gerar um arquivo comprimido de menor tamanho que possa, sem perdas, ser descomprimido futuramente para o seu estado original. Além disso, deve-se utilizar uma **Trie Compacta** que ajude a guardar e pesquisar pelos padrões identificados ao longo da execução do programa.

3 O Programa

3.1 Recursos para o desenvolvimento

O programa foi desenvolvido na linguagem **C++** utilizando o editor de texto **Sublime Text**.

Para controle de versionamento e armazenamento do trabalho foi utilizado o GitHub (disponível em: https://github.com/Pendulun/CompressorZL78).

Para a verificação de vazamentos de memória foi utilizado o **Valgrind**.

Para compilação foi utilizado o **g++** e o **Ubuntu 18.04 LTS** para **Windows 10** para conseguir executar o valgrind.

A máquina utilizada para o desenvolvimento, execução e recuperação do tempo de execução dos testes possui 8Gb de memória RAM e processador Intel Core i5-7200U 2.5 GHz.

3.2 Padrão de entrada e saída

De acordo com a especificação do trabalho, o programa deve ser capaz de interpretar argumentos/comandos passados na hora de sua execução.

Para **compressão** de arquivos, o comando é “-c” seguido do nome do arquivo de extensão “.txt”. Exemplo: ./tp1 -c arquivo.txt . Nesse caso, deverá ser gerado um arquivo de nome “arquivo.z78” que representa o arquivo compactado. Importante notar a extensão desse novo arquivo “z78”.

Para **descompressão** de arquivos, o comando é “-x” seguido do nome do arquivo de extensão “.z78”. Exemplo: ./tp1 -x arquivo.z78 . Nesse caso, deverá ser gerado um arquivo de nome “arquivo.txt” que representa o arquivo descomprimido que deve ser idêntico ao original.

Além desses dois comandos, um terceiro comando opcional poderá ser passado: “-o”. Esse comando serve para indicar explicitamente o nome do arquivo de saída em ambas as situações: compressão e descompressão. Exemplos:

./tp1 -c arquivo.txt -o arquivoComprimido.z78

./tp1 -x arquivo.z78 -o arquivoDescomprimido.txt

3.3 Divisão de pastas e arquivos

O programa foi dividido em quatro pastas: **src**, **include**, **build** e **testcases**. Na pasta **src** estão todos os arquivos **.cpp**, na pasta **include** estão todos os arquivos **.hpp**, na pasta **build** estão todos os arquivos **.o** e na pasta **testcases** estão os arquivos usados para testar a compressão. Além disso, existe o **makefile** para compilação do programa.

3.4 Compilação do Programa

Para compilar o programa, como dito anteriormente, existe o arquivo makefile. Nesse arquivo, está especificado na variável **EXEC** que o nome e extensão do programa gerado será igual a “./tp1” . Nesse caso, o arquivo não terá extensão e não será possível utilizá-lo em um sistema operacional Windows. Já em Linux, para rodar o programa poderemos simplesmente seguir os exemplos da seção 3.2.

Para gerar um arquivo executável, deve-se acrescentar na mesma variável EXEC a extensão “.exe”. Exemplo: “EXEC = ./tp1.exe” .

Depois dessas configurações, basta executar o comando “make” na pasta raiz do programa.

3.5 Lógica da implementação

3.5.1 Compressão

Seguindo a lógica do algoritmo LZ78, deve-se guardar os diferentes prefixos presentes no texto em um dicionário que liga esse dito prefixo com um código aqui chamado de índice.

O algoritmo funciona da seguinte maneira:

1. Adicione um par prefixo: “” (vazio) e índice: 0 no dicionário;
2. Defina o padrão P lido até agora como “” (vazio) e um índice E como 0 e índice total Y como 0;
3. Leia a próxima letra L1 do texto;
4. Caso a combinação P+L1 não esteja presente no dicionário, adicione um novo par prefixo:L1 índice: E no dicionário e incremente Y por 1
5. Caso esteja presente, defina P = P + L1 e guarde em E o índice I referente ao par aonde foi achado a combinação P+L1
6. Repita o passo 3 até acabar o texto

Dessa forma, pode-se detectar as principais atividades a serem realizadas no programa: Pesquisar um padrão no dicionário e adicionar um padrão no dicionário. É nessa hora que a Trie Compacta entra em ação. Com a Trie, será possível procurar por um padrão e adicionar um novo de maneira eficiente. A Trie agirá como o dicionário ao possuir nós que associam o padrão que ele representa com o seu código/índice.

Da forma que o algoritmo funciona, um padrão não estará na Trie caso a leitura da próxima letra do texto combinada com o padrão já existente no dicionário que foi lido até o momento, resulta em um padrão novo. Desse modo, o novo padrão se diferencia do já existente por uma letra.

Como é uma Trie Compacta, apenas nós folhas devem representar um padrão e terem seus valores/índices levados em consideração. Assim, para adicionar um padrão novo na Trie, deve-se, primeiramente, encontrar o nó folha i que representa um padrão x no qual o novo padrão se diferencia. Depois, deve-se adicionar um novo filho em i que representa o próprio padrão de i e um novo nó que representa o novo padrão lido. A Figura 1 exemplifica essa operação:

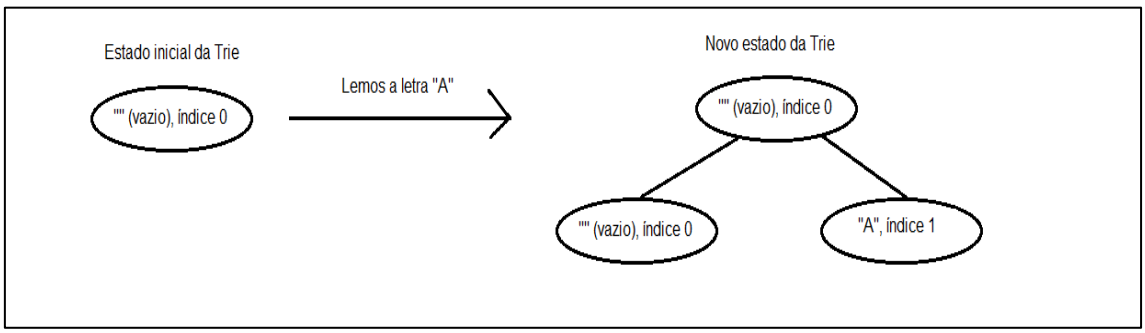

Figura 1: Primeira inserção na Trie

E, caso o texto seguinte fosse composto das letras “AB”, encontraríamos o padrão “A” na árvore, acumularíamos essa letra “A” no padrão lido até agora e verificaríamos se, ao ler o “B”, o padrão “AB” existe na árvore. Nesse caso, o padrão não está presente e deverá ser adicionado na Trie. A Figura 2 exemplifica esse caso:

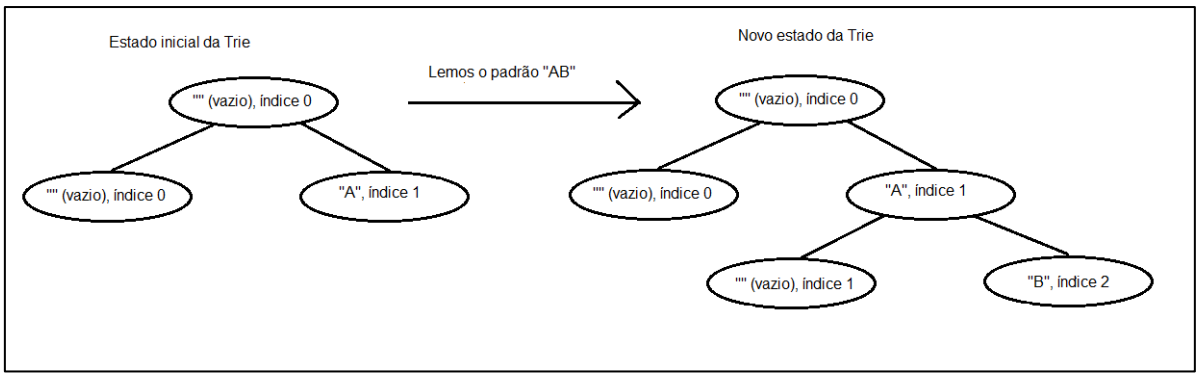

Figura 2: Segunda inserção na Trie

E assim, sucessivamente, encontrando novos padrões no texto e os adicionando na Trie.

Pode-se perceber que cada nó terá apenas uma letra associada a ele que é a letra que foi lida para o padrão representado se diferenciar do seu nó pai.

Ao longo da criação da Trie, são guardados os códigos que deverão ser lidos para a descompressão do arquivo. Esses códigos seguem o seguinte padrão:

- (índice do nó pai do padrão representado agora, letra que diferencia o nó filho do nó pai)

3.5.2 Descompressão

Dado o arquivo comprimido, vamos voltar o texto do arquivo original de acordo com as tuplas gravadas. Depois de ler a forma binária que foram salvos na compressão e a transformar para uma forma que seja possível interpretar os seus valores, vamos adicionando as tuplas, uma a uma, em um vetor. Dessa forma, seremos capazes de reimprimir o texto com um algoritmo dinâmico. O modo que o algoritmo funciona é exemplificado na Figura 3:

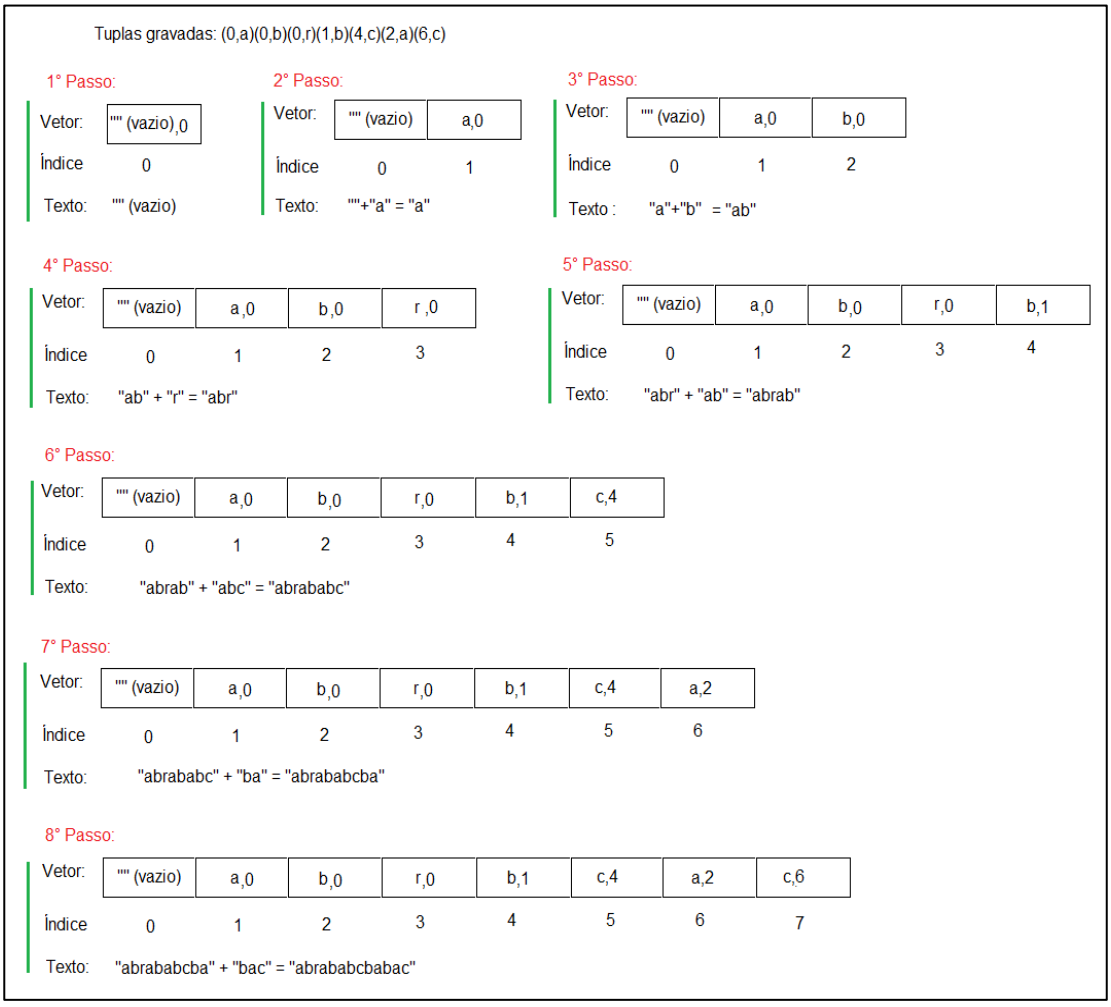

Figura 3: Descompressão de arquivo

Pode-se perceber que o algoritmo lê as tuplas uma a uma e as salva em um vetor. O índice que vem em uma tupla é acessado recursivamente para descobrir qual o padrão que já existia na árvore antes do novo padrão ser adicionado. Na figura 3, esse comportamento é mostrado pela primeira vez ao se ler a tupla (1,b). Isso significa que o texto a ser escrito no arquivo descomprimido é o “b” precedido da letra no índice 1 do vetor que, no caso, é “a”, totalizando “ab”.

Um caso maior é o de ler a última tupla (6,c). Sabemos que se deve escrever a letra “c” precedida pela letra no índice 6, porém, ao acessar o índice 6, vemos que o “a” deve ser precedido pela letra no índice 2: “b”. No final, temos que adicionar a sequência inteira: ”bac”.

Dessa forma, vamos recriando o texto comprimido de acordo com as tuplas.

3.6 Implementação do programa

Dado as funcionalidades que o programa deve ter, existem três classes principais: CompressorLZ78, TrieEncoder e Node.

3.6.1 Node

A classe Node representa um nó dentro de uma Trie, no nosso caso, TrieEncoder.

Agindo como qualquer outro nó em uma árvore, ela possui uma nomeação, um valor e uma lista de outros nós para que ela aponta.

O principal método nele é o de adicionar um outro nó na sua lista de nós apontados ou filhos. Nesse método, ele deve ser capaz de gerar, primeiro, caso for folha, o seu nó filho que representa o seu próprio valor antes de inserir o outro nó como filho. Desse modo, um nó só poderá possuir 0 ou mais de 1 filho.

3.6.2 TrieEncoder

A classe TrieEncoder representa a Trie e possui um nó raiz. Além disso, ela mantém salvo o maior índice de um nó salvo na Trie.

Os seus principais métodos são o de pesquisa de um padrão, que retorna ou o nó no qual, a partir dele, deverá ser adicionado um novo nó que possui um novo padrão, ou o nó que representa o nó em que foi achado o padrão pesquisado, e o método de inserção de um nó que recebe o nó pai e o nó filho que deverá ser adicionado à lista de nós filhos do nó pai.

3.6.3 CompressorLZ78

Essa classe tem as funcionalidades de comprimir e de descomprimir um arquivo. Ela possui os dois métodos que realizam essas ações de acordo com a lógica já explicada nas seções 3.5.1 e 3.5.2.

4 Testes realizados

Para realização da medição do ganho em tamanho de arquivo, foram feitas 31 compressões em livros texto disponibilizados no site https://www.gutenberg.org/ .

Para cada livro, foi comparado o tamanho do arquivo original com o tamanho do arquivo compactado e, com base nessa diferença, foi calculado a porcentagem de espaço salvo da seguinte forma:

$$\left(1 - \left(\frac{tamanhoCompactado}{tamanhoOriginal}\right)\right) \times 100$$

Assim, uma porcentagem será gerada.

Segue os resultados obtidos:

NomeArquivo	Tamanho (KB)		Espaço Salvo (%)
	Original	Comprimido	
aliceWonderland	171	163	4,678362573
annaEkaterina	2020	1437	28,86138614
anneGreenGables	593	496	16,35750422
beowulf	295	261	11,52542373
communistManiphest	93	94	-1,075268817
crusoe	639	517	19,09233177
davidCopperfield	1986	1446	27,19033233
dracula	863	700	18,88760139
frankenstein	455	397	12,74725275
guliver	600	501	16,5
HoD	232	224	3,448275862
hound	380	310	18,42105263
huckleberyFinn	603	499	17,24709784
illiada	1131	908	19,71706454
Leviatan	1226	916	25,28548124
littleWoman	1029	825	19,82507289
metamorphosis	139	136	2,158273381
mobyDick	1247	1011	18,92542101
monteCristo	558	467	16,30824373
odyssey	702	575	18,09116809

oPrincipe	302	273	9,602649007
peterPan	284	258	9,154929577
pridePrejudice	781	578	25,99231754
sherlock	594	496	16,4983165
theRepublic	1211	905	25,26837325
tomCabin	1055	841	20,28436019
tomSawyer	423	375	11,34751773
ulysses	1550	1281	17,35483871
warWorlds	359	323	10,02785515
wizardOz	232	208	10,34482759
zarathustra	665	565	15,03759398
		Média	15,64856956

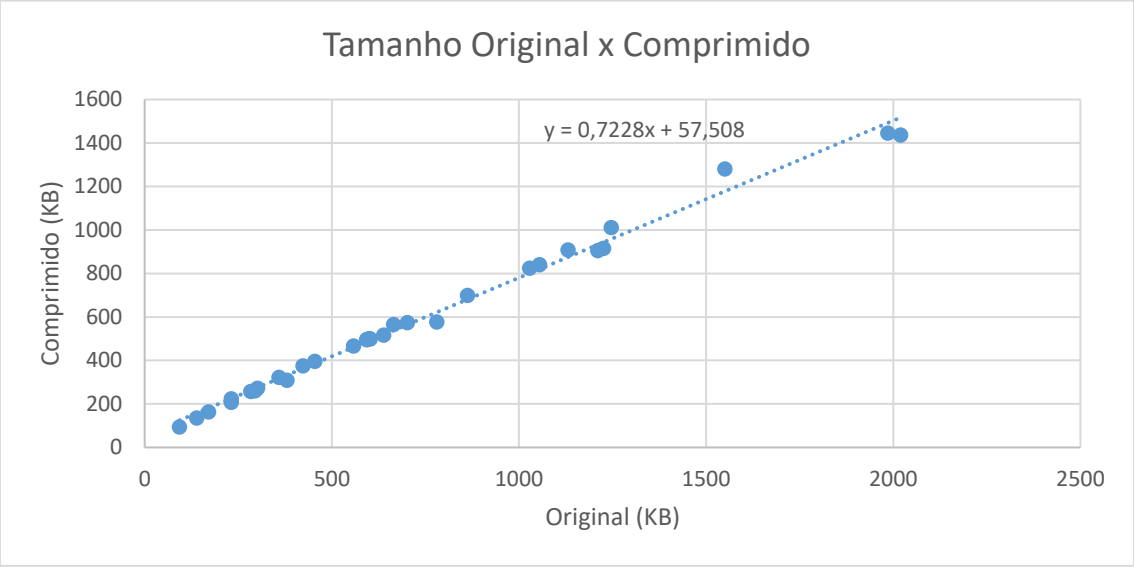

Figura 4: Comparação do tamanho original do arquivo e o tamanho final

5 Conclusão

O objetivo do trabalho era implementar um algoritmo de compressão e descompressão de arquivos de acordo com o algoritmo LZ78 e que utilizasse uma Trie Compacta. No final do trabalho, pode-se dizer que esse objetivo foi alcançado.

As maiores dificuldades do trabalho foram, em ordem cronológica:

1. Entender o funcionamento do algoritmo LZ78
2. Implementar a inserção e pesquisa na Trie Compacta
3. Descobrir como inserir e ler arquivos binários

De acordo com os resultados das compressões dos arquivos, foi interessante perceber uma certa linearidade com relação à proporção salva na compressão, conseguindo gerar uma boa reta de aproximação de forma y = 0.7228x + 57.508 sendo x o tamanho inicial e y o final.

Apesar disso, no arquivo communistManiphest, que, inicialmente possuía 93 KB, ao ser comprimido, acabou ganhando 1 KB de tamanho. Isso pode ser um sinal de que, quanto menor o tamanho do arquivo e o número de padrões encontrados, as informações inseridas no arquivo comprimido serão maiores do que o original.

6 Referências

- Site: < https://www.gutenberg.org/ >, último acesso em 07/02/2021
- Site: < https://www.cplusplus.com/ >, último acesso em 07/02/2021
- Site: < https://pt.wikipedia.org/wiki/LZ78 >, último acesso em 05/02/2021