Trabalho Prático 1 - Algoritmos 2

Matheus Aquino Motta[1]

[1]Bacharelado em Matemática Computacional
DCC - Universidade Federal de Minas Gerais

matheusaquino199@gmail.com.br 2018046513

Abstract. *In this report we will briefly discuss the implementation of the assignment 1 of the subject algorithms 2. The problem consisted into implementing a LZ78 file compression algorithm, which should properly compress and decompress files within the conditions specified, using a Compressed Trie structure to save the dictionary information from each file. And some results about the data compression ratio will also be presented.*

Resumo. *Nesse relatório iremos discutir brevemente a implementação do trabalho prático 1 da disciplina algoritmos 2. O dado problema consistia na implementação de um compressor de arquivos LZ78, que deveria comprimir e descomprimir arquivos de maneira apropriada, assim como especificado, utilizando uma Trie Compacta como dicionário para armazenar os textos encontrados. Além disso, alguns resultados acerca da taxa de compressão alcançada também serão apresentados.*

1. Introdução

Durante a primeira metade da disciplina de Algoritmos 2 tivemos contato com diversos algoritmos e estruturas de dados para manipulação de sequências. Assim, nesse trabalho apresentaremos a implementação de um compressor de arquivos que faz uso de uma das estruturas discutidas em aula, Trie Compacta, para a execução apropriada do protocolo de compressão LZ78.

De um modo geral o algoritmo irá funcionar seguindo o protocolo explicado na página da wikipédia *https://pt.wikipedia.org/wiki/LZ78Exemplo*. Iremos receber um arquivo de texto que será lido pelo programa caractere-a-caractere, assim, a partir do texto recebido será construído um dicionário que irá armazenar a partir de uma Trie Compacta os prefixos existentes no texto.

2. Implementação

O protocolo de compressão de arquivos LZ78 foi implementado na linguagem C++, e podemos descrever o funcionamento do algoritmo para a compressão e descompressão do arquivo de arquivos da seguinte forma.

2.1. Compressão

Seja T o texto recebido no arquivo e C uma cadeia de caracteres incialmente vazia. Iremos construir o dicionário D analisando cada T_i, isto é caractere-a-caractere, em função de seus prefixos, onde para cada T_i adcionaremos $S = C + T_i$ ao dicionário D e iremos verificar se S é um prefixo já representado na estrutura. Caso seja, apenas redefinimos C

como $C = C+T_i$. Caso o contrário, iremos adicionar atualizar a nossa resposta e redefinir a cadeia de *matchings* C como vazia. Assim, repetimos o processo para o caractere T_{i+1} até o fim do texto T.

O dicionário D, assim como já mencionado, foi implementado como uma Trie Compacta. A trie compacta foi implementada de modo que para cada inserção de palavras procuramos de forma apropriada na estrutura caractere-a-caractere o maior prefixo comum entre a palavra a ser adicionada e os representados pela estrutura. Entretanto, para propósitos do algoritmo, sabemos que pela construção das palavras, ou a palavra é um prefixo já existente no dicionário, isto é, um prefixo de uma palavra anteriormente adicionada, ou é um prefixo já existente acrescido de um novo caractere.

Desse modo, seja S uma palavra construída pelo processo descrito acima, sabemos que ou $S \in D$, ou $S - T_i \in D$. Dessa forma, para o primeiro caso apenas seguimos para o caractere seguinte do texto, enquanto para o segundo inserimos o caractere T_i, em função do seu prefixo $S - T_i$, em D de 3 diferentes formas:

- Caso o último caractere em que um *matching* de prefixos ocorreu seja relativo a um nó diferente da raiz e sem filhos, apenas adicionamos o caractere ao nó, assim como seu respectivo id.

- Caso o último caractere em que um *matching* de prefixos ocorreu tenha filhos ou seja a raiz, adicionamos um novo nó filho a ele, com o caractere relativo ao *mismatching* e seu respectivo id.

- Caso o último caractere em que um *matching* de prefixos ocorreu seja um caractere intermediário de algum nó, devemos particionar o nó em dois, um relativo ao prefixo de S e o existente na estrutura, e um do sufixo que não possui o caractere T_i relativo ao *mismatching*. Assim, o nó atual passa a armazenar informações sobre o prefixo comum, e irá ter como filhos, um nó para o caractere relativo ao *mismatching* T_i, e outro para o sufixo, mantendo informação relativa a toda a sub-árvore potencialmente já existente.

Assim, para cada inserção realizada no dicionário armazenamos uma tupla (id, T_i) a nossa resposta relativa ao caractere T_i adicionado, e o id do caractere "pai" de T_i na árvore. O conjunto de tuplas encontrado ao final é o resultado final da compressão do arquivo, haja vista que iremos possuir toda informação necessária para a reconstrução do texto do arquivo a partir dele por meio da descompressão. Podemos visualizar a compressão a partir da construção do dicionário D, a partir do exemplo *T = A_ASA_DA_CASA*.

Inicializamos nossa cadeia como inicialmente vazia $C = ""$ e inicializamos o dicionário D com um nó relativo a um texto vazio com id 0.

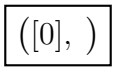

Figura 2.1.1: Nó inicial raiz do dicionario.

Assim, tomamos $S = C + T_0 =$ 'A' e adicionamos S ao dicionário, onde ao percorrer a árvore verificamos que estamos no caso de inserção 2, e criamos um novo nó para o caractere "A" de $id = 1$.

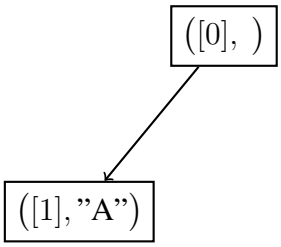

Figura 2.1.2: Adição da string $S =$ "" $+ T_0$ na estrutura.

E como especificado, ao final, como realizamos a inserção de um novo caractere, a tupla $(0,$ 'A') é adicionada à resposta. Para o caractere $T_1 =$'_' o processo se repete de forma identica ao ocorrido para T_0 e adicionamos a tupla $(0,$ '_') na resposta.

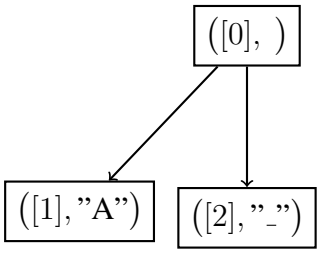

Figura 2.1.3: Adição da string $S =$ "" $+ T_1$ na estrutura.

Quando recebemos o próximo caracter $T_2 =$'A', temos $S =$'A', que é um prefixo já existente no dicionário, desse modo não precisamos atualizar a nossa Trie e redefinimos C como $C = C + T_2$.

Para $T_3 =$'S', teremos uma cadeia $C =$"A" e portanto $S =$"AS", assim, ao inserirmos S em D encontramos um prefixo "A" igual ao de S e um *mismatching* no caractere 'S', relativo a T_3. Desse modo, caimos no primeiro caso de inserção e atualizamos a Trie.

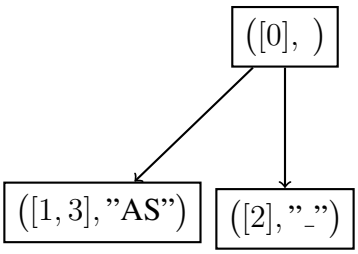

Figura 2.1.4: Adição da string $S =$ "A" $+ T_3$ na estrutura.

Adicionando a tupla $(1,$ 'S') a resposta e redefinindo C como vazia novamente.

Agora com a cadeia novamente vazia temos que T_4 para T_4, uma palavra $S =$"A" será inserida, que por sua vez é um prefixo já existente e nada será alterado no dicionário, e C será atualizada para $C =$"" + 'A'. Com uma cadeia $C =$"A", temos para a inserção de

S =”A”+T_5 =”A_” no dicionário que um *mismatching* irá ocorrer e uma inserção será realizada de acordo com o terceiro caso.

Desse modo, quebramos o nó em função do prefixo e sufixo da posição em que o *mismatching* ocorreu e atualizamos a árvore. Adicionando na resposta a tupla $(1, \text{'\_'})$.

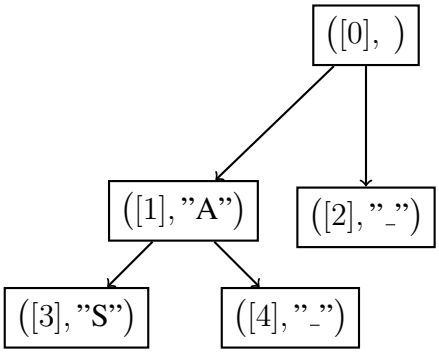

Figura 2.1.5: Adição da string $S =$ ”A” $+ T_5$ na estrutura.

Novamente, para T_6 com a C vazia caimos no primeiro caso, atualizamos a árvore e adicionamos na resposta a tupla $(0, \text{'D'})$.

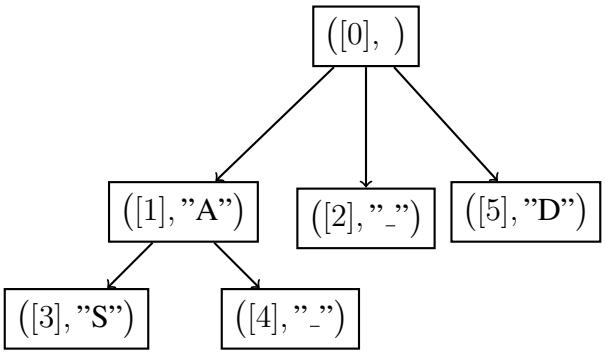

Figura 2.1.6: Adição da string $S =$ ”A” $+ T_6$ na estrutura.

Para os caracteres T_7 e T_8, iremos inserir palavras S que já estão contidas no dicionário, e nada será alterado na estrutura. Agora para T_9 ='C' e uma cadeia formada por T_7 e T_8, teremos uma palavra S =”A_C”, e iremos realizar a inserção de acordo com primeiro caso, adicionando a resposta a tupla $(4, \text{'C'})$.

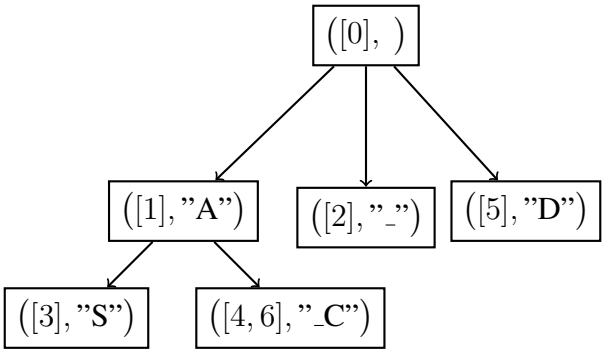

Figura 2.1.7: Adição da string $S =$ "A_" $+ T_9$ na estrutura.

Com a cadeia novamente vazia após a inserção recebemos os caracteres T_{10} e posteriormente T_{11} que irão gerar palavras S que são prefixos de palavras já inseridas no dicionário. Desse modo, por fim, para T_{12} com uma cadeia $C =$"AS", iremos inserir a palavra $S =$"ASA" e iremos realizar uma inserção relativa ao caractere T_{12} de acordo com o caso 1.

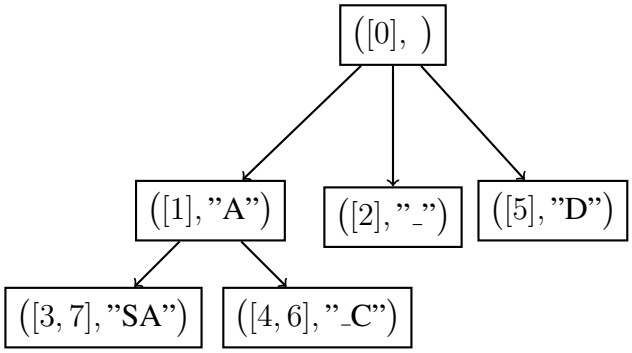

Figura 2.1.6: Adição da string $S =$ "AS" $+ T_{12}$ na estrutura.

Assim, adicionando a tupla $(3, \text{'A'})$ na resposta, obtemos o resultado final da construção do dicionário e consequente compressão do arquivo a partir da sequência de inserções obtida (0,'A')(0,'_')(1,'S')(1,'_')(0,'D')(4,'C')(3,'A').

Caso a busca termine com uma cadeia estritamente igual algum prefixo do dicionário uma tupla é adicionada para que a descompressão seja realizada apropriadamente.

Essa sequência de tuplas é salva em código binário em um arquivo, ocupando 4 bytes para cada inteiro e 1 byte para cada char, sem uso dos caracteres separadores.

2.2. Descompressão

A descompressão foi realizada a partir da leitura apropriada do arquivo binário gerado, onde para cada inteiro i e char ch obtido reconstruimos nosso arquivo iterativamente de acordo com as cadeias C adicionadas durante a compressão.

Inicialmente para cada tupla inicializamos $C = ch$ e por meio de um hashing na tupla do id "pai", concatenamos em C o caractere relativo a ele em sua cadeia, atualizando o id "pai" a cada concatenação, até que o id "pai" seja igual a 0, isto é, a raiz.

Podemos visualizar esse processo como um *backtracking* na árvore, onde cada tupla visitada é uma folha em relação as tuplas já lidas. Assim, partindo de cada folha obtemos a cadeia *C* invertida relativa a sua tupla no arquivo de compressão.

Desse modo, invertendo a cadeia obtida e adicionando-a à resposta cada iteração obtemos nosso arquivo original de forma apropriada.

3. Análise de complexidade

Seja T o texto a ser comprimido, temos que a complexidade de tempo de execução do algoritmo para compressão é da ordem de $O(|T|k)$, onde k é o maior prefixo existente na Trie Compacta, desse modo no pior caso iremos obter uma complexidade de tempo para compressão da ordem de $O(|T|^2)$. Esse resultado pode ser facilmente visualizado, haja vista que para cada caractere do texto percorremos a Trie Compacta uma vez, onde no pior caso iremos percorrer sempre o maior prefixo, isto é, a altura da árvore.

Para a descompressão do arquivo de forma análoga temos uma complexidade de tempo da ordem de (nk), onde n é o número de tuplas e k o maior número de operações necessários para se chegar do id inicial da tupla ao id da raiz, isto é, a distância da folha à raiz, altura da árvore.

A complexidade de espaço adicional necessária para compressão é da ordem de $O(|T| + n)$, haja vista que é o espaço ocupado pelo texto no dicionário e das tuplas sendo iterativamente construídas.

Enquanto para descompressão temos uma complexidade da ordem de $O(n+|T|)$, onde n é número de tuplas ocupado pelo *hash* utilizado e $|T|$ o tamanho do texto sendo obtido iterativamente a partir das cadeias.

4. Instruções de compilação e execução

Foram entregues os arquivos *ċpp* e *ḣ* relativos a implementação do algoritmo, assim como um makefile, e 10 arquivos teste.

Para a execução do algoritmo basta a utilização do comando *make* no diretório com todos os demais arquivos e posterior execução para compressão e descompressão respectivamente como:

./main -c input_file -o output_file
./main -x input_file -o output_file

Caso um arquivo output não seja dado, o programa irá salvar o resultado em um arquivo de nome relativo ao do input.

Os arquivos teste disponibilizados são livros baixados do *Project Gutenberg*, todos em *plain text*, .txt.

5. Resultados

O algoritmo foi capaz de comprimir e descomprimir arquivos de forma apropriada, onde comparando os arquivos iniciais utilizados para a compressão e os seus respectivos resultados após a descompressão foram obtidos arquivos identicos.

	Tamanho Original	Tamanho Comprimido	Taxa de Compressão		
in_0.txt	157.1 KB	156.8 KB	0.0019		
in_1.txt	1.9 MB	1.4 MB	0.2631		
in_2.txt	514.4 KB	444.9 KB	0.1351		
in_3.txt	184.6 KB	180.8 KB	0.0205		
in_4.txt	138.1 KB	134.9 KB	0.0231		
in_5.txt	1.2 MB	994.5 KB	0.1712		
in_6.txt	581 KB	492.3 KB	0.1526		
in_7.txt	781 KB	577.6 KB	0.2604		
in_8.txt	451.3 KB	393.2 KB	0.1287		
in_9.txt	423 KB	374.3 KB	0.1151		

Table 1. Tabela de taxa de compressão dos arquivos teste.

Podemos visualizar as taxas de compressão de dados obtida pelo protocolo em relação aos exemplos disponibilizados a partir da tabela.

Onde mesmo alguns não obtendo altas taxas de compressão como o teste 0 e 4, equanto outros apresentaram resultados mais expressivos e redução de cerca de 26% do tamanho original do arquivo como os testes 2 e 7. Essas variações podem ser explicadas devido ao fato que a compressão do arquivo baseia-se na existência de cadeias/prefixos igual no texto, o que não pode ser garantido para todos.

6. Conclusões

Durante a implementação do trabalho prático 1 tivemos a oportunidade de lidar com estruturas e algoritmos de manipulação de sequência com arquivos reais, o que foi uma ótima oportunidade para aprender a como lidar com tais problemas em cenários não ótimos.

A construção da Trie Compacta foi um desafio devido ao leque de exemplos de implementaçõs disponíveis para estudo e necessidade de implementar o algoritmo diretamente de sua perspectiva teórica. Entretanto os resultados demonstraram-se extremamente positivos e a execução foi feita apropriadamente.

A descompressão foi realizada utilizando um hashing para ganho de eficiência, entretanto essa poderia também ter sido realizada com o uso da construção iterativa de uma Trie Compacta a partir das tuplas obtidas.