

# TP1 Algoritmos 2 - DCC207

Lucas Paulo Martins Mariz

- **Objetivos**

Nesse trabalho foi proposto a exploração, de forma prática, dos conceitos de manipulação de sequências principalmente no que tange a compressão de arquivos. Mais especificamente, o objetivo principal se apresentou na implementação do método LZ78 juntamente à árvores de prefixos.

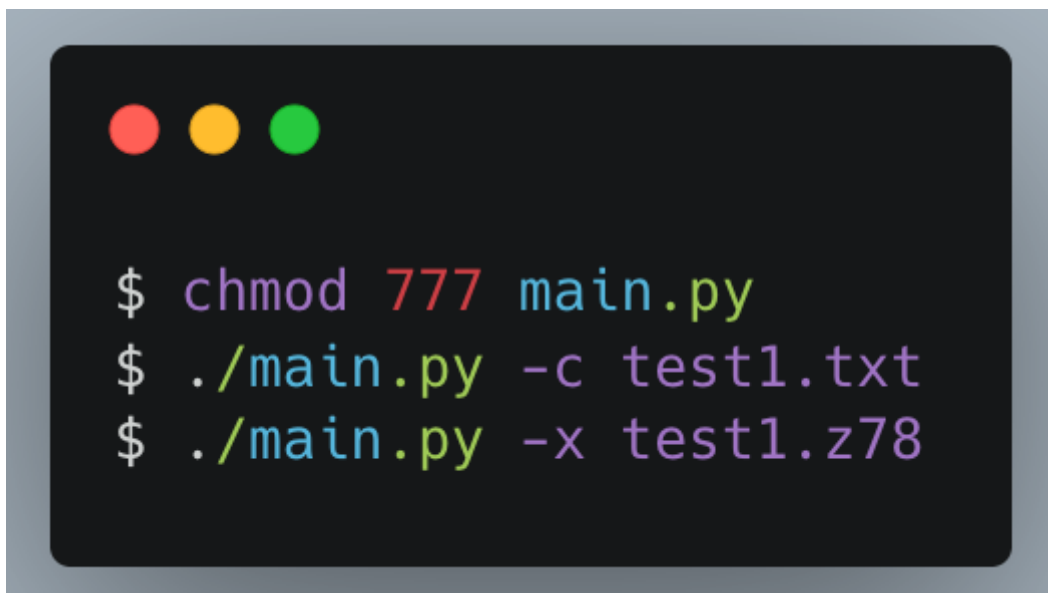
- **Instruções**

O repositório do github referente ao trabalho pode ser encontrado em <https://github.com/LucasPMM/lz78-compression>

A linguagem python foi escolhida para a realização do trabalho. Entre muitos fatores, a facilidade de utilização, bem como sua versatilidade, foram alguns dos critérios utilizados nessa escolha.

A primeira execução do programa, por sua vez, pode apresentar alguns erros de permissão. Para contorná-los é bem simples, basta seguir o seguinte fluxo de comandos.

OBS: O comando "chmod" dá as permissões necessárias ao script para que o mesmo possa ser executado.

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. It displays three terminal commands in a monospaced font with syntax highlighting: the first command sets permissions to 777 for main.py, the second runs the script with a test file, and the third runs the script to produce a compressed file.

```
$ chmod 777 main.py
$ ./main.py -c test1.txt
$ ./main.py -x test1.z78
```

- **Implementação**

Primeiramente, foi utilizada uma trie como estrutura principal de manipulação. Sua definição consiste, basicamente, em um construtor, um método para pegar o prefixo de uma palavra, um método para checar se uma palavra está presente na trie e um método para adicionar elementos à estrutura. Sua implementação pode ser vista no pseudocódigo abaixo:

```
class Trie:
    # Trie constructor
    def __init__(self, value='', numeric_value=None, parent=None, empty=True):
        (...)

    # Get prefix of a word
    def _get_prefix(self, word):
        (...)

    # Returns if the trie contains the word
    def contains(self, word):
        (...)

    # Add one value to the trie
    def add(self, word, value):
        (...)
```

A compressão se baseia na representação do texto como um conjunto de prefixos ordenados, na qual cada sequência é identificada como um prefixo anteriormente cadastrado mais um caractere novo, caso necessário.

A utilização da trie nesse processo é fundamental, pois, com sua estrutura, podemos economizar espaço no armazenamento de prefixos. Em mapeamentos tradicionais, prefixos iguais poderiam ser duplicados ao longo do dicionário, fato que não ocorre na estrutura implementada, na qual a forma de indexação dos prefixos diminui tal ambiguidade.

O fluxo de compressão segue a seguinte lógica: passamos por todo o arquivo e checamos se a cadeia com o caractere já existe nos prefixos. Se sim, adicionamos o caractere a cadeia e continuamos para o próximo caractere do texto. Se não, adicionamos o novo prefixo e escrevemos no arquivo comprimido, o index é atualizado e passamos para o próximo caractere. O código abaixo representa a lógica implantada:

```
import trie as T
# Global variables
import glv

def compress(input_file, output_file):
    with open(input_file, 'r') as input:
        text = input.read()
    with open(output_file, 'wb') as output:
        index = 1
        chain = ''
        trie = T.Trie('', numeric_value=0)

        for c in text:
            contains = trie.contains(chain + c)
            if contains != -1:
                chain += c
                continue

            chain_index = trie.contains(chain)
            trie.add(chain + c, index)
            output.write(chain_index.to_bytes(glv.INT_SIZE, 'big'))
            output.write(ord(c).to_bytes(glv.CHAR_SIZE, 'big'))
            chain = ''
            index += 1
```

A descompressão, por sua vez, ocorre de forma um pouco mais simples. A lógica consiste em passar por todo o arquivo comprimido, identificá-lo na trie e o adicionar na cadeia de descomprimido, a qual será, após chegar no fim do arquivo, escrita no arquivo de saída. O código abaixo representa a lógica implantada:



```

import utils
# Global variables
import glv

def decompress(input_file, output_file):
    with open(input_file, 'rb') as file:
        decoded = ''
        trie = ['']
        while True:
            index = file.read(glv.INT_SIZE)
            if index == b'':
                break

            index = int.from_bytes(index, "big")
            char = file.read(glv.CHAR_SIZE)
            if char == b'':
                char = ''
            else:
                char = chr(int.from_bytes(char, 'big'))

            chain = trie[index]
            new_chain = chain + char
            decoded += new_chain
            trie += [new_chain]
        utils.write_file(output_file, decoded)

```

## • Testes

Foram realizados diversos testes em arquivos de tamanhos variados com o intuito de avaliar a taxa de compressão. Vamos dividir os testes em 2 partes, uma com maior **precisão** e outra com maior **eficiência**. Essa diferenciação foi feita devido a uma variação na função "contains" da trie, a qual pode ou não checar "caracteres vazios". Com as duas linhas comentadas temos uma compressão / descompressão 100% precisa, ou seja, ao descomprimir o arquivo comprimido ele será idêntico àquele que foi comprimido. Por outro lado, se descomentarmos as linhas assinaladas, temos que a compressão se torna muito mais eficiente, chegando a reduzir o arquivo pela metade, mas eventuais perdas de caracteres

podem ocorrer (não é comum, mas ocorreu em 1 dos 20 testes feitos). Assim, vamos analisar os dois cenários.

```
def contains(self, word):
    # if word == '' and '' in self.children.keys():
    #     return self.children[''].numeric_value

    if word == self.value and self.numeric_value != None:
        return self.numeric_value

    index, key = self._get_prefix(word)
    if key == None:
        return -1
    return self.children[key].contains(word[index:])
```

**Caso Eficiente:** nesses testes, tal taxa se apresentou diretamente relacionada ao tamanho do arquivo, bem como seu formato. Além disso, quanto maior o arquivo, maior foi o tempo de execução do script, sendo praticamente instantâneo nos melhores casos ou demorando até 30 segundos nos piores.

Podemos conferir abaixo o resultado da compressão em alguns arquivos de tamanhos variados (1KB até 2MB). A taxa de compressão funciona da seguinte forma: caso positiva significa que o arquivo cresce (no arquivo test1.txt o arquivo cresceu 78.49%). Por outro lado, se a taxa for negativa o arquivo diminuiu de tamanho (no arquivo test10.txt o arquivo diminuiu 53.80%).

**Tabela 1:**

**Tabela 2:**

<pre> Compression: test1.txt -&gt; test1.z78 Input 1004 bytes Output 1792 bytes Compression ratio +78.49%  Compression: test2.txt -&gt; test2.z78 Input 5020 bytes Output 6484 bytes Compression ratio +29.16%  Compression: test3.txt -&gt; test3.z78 Input 10040 bytes Output 11008 bytes Compression ratio +9.64%  Compression: test4.txt -&gt; test4.z78 Input 50162 bytes Output 43268 bytes Compression ratio -13.74%  Compression: test5.txt -&gt; test5.z78 Input 100324 bytes Output 74156 bytes Compression ratio -26.08%  Compression: test6.txt -&gt; test6.z78 Input 200648 bytes Output 129836 bytes Compression ratio -35.29%  Compression: test7.txt -&gt; test7.z78 Input 501620 bytes Output 279404 bytes Compression ratio -44.3%  Compression: test8.txt -&gt; test8.z78 Input 1003240 bytes Output 502368 bytes Compression ratio -49.93%  Compression: test9.txt -&gt; test9.z78 Input 1504860 bytes Output 705752 bytes Compression ratio -53.1%  Compression: test10.txt -&gt; test10.z78 Input 2006480 bytes Output 927028 bytes Compression ratio -53.8%</pre>	<pre> Compression: test1.txt -&gt; test1.z78 Input 1001 bytes Output 4004 bytes Compression ratio 300.0%  Compression: test2.txt -&gt; test2.z78 Input 5018 bytes Output 20072 bytes Compression ratio 300.0%  Compression: test3.txt -&gt; test3.z78 Input 10038 bytes Output 40152 bytes Compression ratio 300.0%  Compression: test4.txt -&gt; test4.z78 Input 50156 bytes Output 200624 bytes Compression ratio 300.0%  Compression: test5.txt -&gt; test5.z78 Input 100322 bytes Output 401288 bytes Compression ratio 300.0%  Compression: test6.txt -&gt; test6.z78 Input 200642 bytes Output 802568 bytes Compression ratio 300.0%  Compression: test7.txt -&gt; test7.z78 Input 501615 bytes Output 2006460 bytes Compression ratio 300.0%  Compression: test8.txt -&gt; test8.z78 Input 1003235 bytes Output 4012940 bytes Compression ratio 300.0%  Compression: test9.txt -&gt; test9.z78 Input 1504858 bytes Output 6019432 bytes Compression ratio 300.0%  Compression: test10.txt -&gt; test10.z78 Input 2006478 bytes Output 8025912 bytes Compression ratio 300.0%</pre>
--	---

**Caso Preciso:** os testes nesse cenário foram bem monótonos, sendo que o arquivo cresceu em todos eles na mesma taxa. Os valores exatos podem ser conferidos na tabela 2.

Os testes relacionados à descompressão foram omitidos mas seguem a mesma linha de raciocínio, sendo que os arquivos comprimidos voltam a ter o extamente o tamanho original (antes de serem comprimidos).

## ● Conclusão

Após a finalização do trabalho e realização de diversos testes é possível retirar algumas conclusões.

O método LZ78 não é ótimo para todos os textos. Em particular, textos pequenos, sem muitas repetições, tendem a ficarem maiores após comprimidos. Por outro lado, sua eficiência aumenta significativamente em determinados casos, em arquivos maiores onde a repetição predomina.

Em relação a complexidade, temos que as operações na trie giram em torno de  $O(n)$ . Por outro lado, o processo de compressão/descompressão passa por todo o texto, o que levaria a uma complexidade  $O(m)$ .

Em suma, o tema proposto foi de grande importância para que conceitos importantes da disciplina pudessem ser fixados de forma mais incisiva. Além disso, tal atividade se apresentou como uma forma única de praticar o conteúdo ensinado.