Trabalho Prático I - Algoritmos II

Bruno Flister Viana - 2018048567

May 2023

1 Introdução

Nesse trabalho serão abordados os aspectos práticos de manipulação de sequências. Especificamente, serão explorados aspectos de implementação de árvores de prefixo aplicada ao problema de compressão de arquivos.
Foi implementado um algoritmo para resolver o problema de comprimir arquivos texto através do método LZ78. Esse método é baseado em dicionários e, basicamente, substitui strings que se repetem no texto por códigos. Como o algoritmo executa muitas buscas e inserções nesse dicionário, é comum utilizar árvores de prefixo na sua implementação. O LZ78, apesar de simples, é a base de vários algoritmos implementados nas ferramentas de compressão utilizadas atualmente.
Nesta documentação temos uma breve descrição do algoritmo, seguido da descrição das funções que o implementam. Por último é realizado uma análise empírica da eficiência do algoritmo.

2 Desenvolvimento

2.1 Metodologia

Para a implementação do algoritmo exposto foi escolhido a linguagem Python 3, utilizando-se apenas de estrutura de dados e funções nativas.
O método LZ78 usa um dicionário para armazenar as sequências de caracteres encontradas no arquivo a ser comprimido. Cada sequência de caractere é acompanhado de um código, representado por um número inteiro, que indica a posição da sequência no dicionário. Ao ler um caractere do arquivo, procura-se no dicionário se o mesmo já se encontra nele. Caso seja encontrado, é lido o próximo caractere, concatenando-o com o anterior e buscando novamente a sequência no dicionário. Enquanto a sequência está presente no dicionário, o algoritmo continua concatenando e verificando. Se é encontrado um caractere c do qual, concatenando com a *string*, não pertence ao dicionário, emitimos na saída o par $(string_{codigo}, c)$, onde $string_{codigo}$ é o código no dicionário da última *string* antes de ser concatenada com c. Em seguida, insere-se o c no dicionário e reinicia-se o algoritmo.
Inicialmente foi implementado a estrutura de dados Trie para a utilização da

mesma como um dicionário para a codificação do texto. Em seguida foi programado e testado a lógica da compressão e descompressão dos dados. Por último, adaptou-se o código para a entrada e saída de dados por arquivos, convertendo-os para bytes com a finalidade de calcular a taxa de compressão.

2.2 Implementação

A implementação tem dois arquivos, sendo um deles (*trie.py*) apenas para armazenar a classe da Trie, que servirá como dicionário para o algoritmo de compressão. O outro arquivo (*main.py*), o principal, armazena a função principal tal como a implementação dos métodos de compressão e descompressão. As funções presentes em cada arquivo estão descritas abaixo.

2.2.1 trie.py

class TrieNode(): Esta é a classe que representa um nodo da estrutura da Trie. Nesta classe temos quatro atributos, sendo eles:

- Inteiro que representa o código da sequência.
- Um tipo char para representar cada elemento da sequência inserida no dicionário.
- Uma variável booleana para indicar o final da sequência, ou seja, indicar um nó folha.
- Um dicionário para armazenar os demais nodos filhos do nodo atual.

class Trie(): A classe da Trie inicia-se com a função que define o nodo raiz. A raiz é indicada pelo código 0 e pela string vazia. Em seguida temos as funções de inserção e busca na Trie, descritas abaixo:

1. **class Trie()::insert():** Para inserir um novo caractere na Trie a função recebe como parâmetro uma palavra e um índice (um inteiro que será passado pelo algoritmo LZ78). Partindo da raiz, a função de inserção verifica para cada caractere da palavra se este já está inserido na Trie, para tal basta olhar o atributo $filhos$ de cada nodo. Se já está inserido, a inserção caminha para o próximo filho e verifica novamente. Se o caractere não existe ainda na Trie, o algoritmo o insere com o índice como código para o caractere atual da palavra. A inserção encerra quando toda a palavra é percorrida. A Trie tem complexidade de espaço na ordem $O(n)$ onde n é o número de elementos da Trie. Note que o tempo de inserção depende do tempo de busca, que também é da ordem $O(n)$.
2. **class Trie()::search():** Esta função procura e retorna o código de uma palavra passada como parâmetro. Novamente partimos da raiz para realizar a busca na árvore e buscamos cada caractere da palavra no dicionário. Verificando se o caractere atual pertence aos filhos do nodo atual, partimos para o próximo nodo onde este caractere está inserido e fazemos

o mesmo para o próximo caractere. Se após percorrer toda a palavra a busca se encontrar num nó folha, indicado pelo atributo booleano da classe dos nodos, retorna-se o código daquele nó. Do contrário, a palavra não existe no dicionário. O tempo de busca depende da quantidade de elementos inseridos na Trie, como estamos sendo percorrendo e verificando os filhos de cada nó, o tempo de busca é da ordem $O(n)$.

2.2.2 main.py

compress(): O método de compressão parte da definição do LZ78. Defini-se o dicionário como uma Trie. Em seguida é feita a leitura do arquivo caractere por caractere. O caractere c atual é concatenado com uma string auxiliar $prefix$. Se $prefix + c$ não está no dicionário, é gravado no arquivo de entrada a tupla $(prefix_{code}, c)$. Caso contrário, concatena as duas strings e segue para o próximo caractere do arquivo texto. A escrita no arquivo de saída converte o índice e o caractere c ambos para bytes de tamanho 3 e 2 respectivamente. Inicialmente foi testado para bytes de tamanho 2 e 1, mas houve problemas quanto a isso que serão discutidos nas análise de testes. A complexidade do algoritmo depende da inserção na Trie e de percorrer o texto do início ao fim, portanto temos $O(n)$, onde n é a quantidade de caracteres do texto.
decompress(): Do mesmo modo ocorre a descompressão, lê-se do arquivo codificado, byte à byte, o índice e o caractere a ele correspondente ; realiza-se a concatenação desses caracteres, usando agora uma lista de *string* como dicionário para armazenar o texto decodificado. Note que a utilização do dicionário como tipo *list* neste caso se deu pela praticidade de se encontrar a palavra a partir do índice, sendo essa operação $O(1)$ para um array em python enquanto a Trie retornaria o a palavra em tempo $O(n)$, como visto anteriormente neste tipo de estrutura de dados. A conversão de bytes ocorre da mesma forma mas na direção oposta desta vez.

2.3 Testes

Os testes realizados foram feitos com dez arquivos retirados do site https://www.gutenberg.org/. Três deles foram fornecidos pelo professor, e os demais correspondem a livros clássicos da literatura mundial. Abaixo está descrito numa tabela cada arquivo, o tamanho de ambos antes da compressão, o tamanho do arquivo comprimido, o tamanho do arquivo após a descompressão e a taxa de compressão de cada um. Nesta tabela está representada os códigos e caracteres comprimidos e representados no arquivo de saída com 3 bytes e 2 bytes, respectivamente.

2.3.1 Taxa de Compressão

É possível notar que a taxa de compressão não passou dos 50% em nenhum caso, de fato, houve uma média de 14.7% da taxa de compressão. Isso se deu muito ao fato da representação em bytes para inteiros utilizar de 3 bytes e para

Name	Size	Compressed	Decompressed	Compress-Rate
alice	171kb	158kb	164kb	7.6%
constituicao1988	637kb	427kb	619kb	32%
domcasmurro	401kb	358kb	402kb	10.7%
dracula	862kb	692kb	861kb	19.7%
littlewoman	1097kb	859kb	1097kb	21.6%
metamorfose	139kb	133kb	138kb	4.3%
odyssey	701kb	564kb	625kb	19.5%
oslusiadas	337kb	314kb	341kb	6.8%
pridenprejudice	755kb	585kb	746kb	22.5%
romeonjuliet	166kb	161kb	164kb	3%

Table 1: Representação em Bytes - Int: 3 bytes ; Char: 2 bytes.

Name	Size	Compressed	Decompressed	Compress-Rate
oslusiadas	337kb	189kb	341kb	43.9%

Table 2: Representação em Bytes - Int: 2 bytes ; Char: 1 bytes.

caracteres utiliza-se 2 bytes. Em testes anteriores notou-se que ao se utilizar representações de 2 bytes e 1 byte, para as respectivas funções, obteve-se uma taxa de compressão bem maior. A comparação se dá com um exemplo acima onde é utilizada essa representação. No entanto, para textos maiores ou com maior variação de palavras, a representação em 2 bytes e 1 byte se mostrou pouca, já que constantemente era necessário usar mais bytes na representação de números maiores. Dos testes, apenas Os Lusíadas admitiu esta compressão. Portanto, optou-se por utilizar 3 e 2 bytes, já que a compressão foi realizada sem os problemas acima citados para todos os testes.

2.3.2 Descompressão

A descompressão foi satisfatória para a maioria dos casos, sendo o texto resultado desta quase idêntico ao arquivo original. Pode-se observar tal fato ao notar que o tamanho de ambos os arquivos são quase idênticos pela **Tabela 1**. Por outro lado, houve alguns problemas na codificação de certos tipos de caracteres, dos quais estão fora do escopo da função de leitura utilizada pelo código. Um exemplo é o arquivo Odyssey. Note que a descompressão total do mesmo não foi possível pois a conversão do tipo em bytes para um caractere presente no arquivo original não era um tipo suportado pela função em Python. Mas grande parte dos textos, que usam caracteres simples em 'utf-8', o programa executou sem problemas, não havendo portanto a necessidade de alteração inicial.
Abaixo está uma comparação dos arquivos original e a descompressão, respectivamente, de um trecho texto do livro Alice no País das Maravilhas.

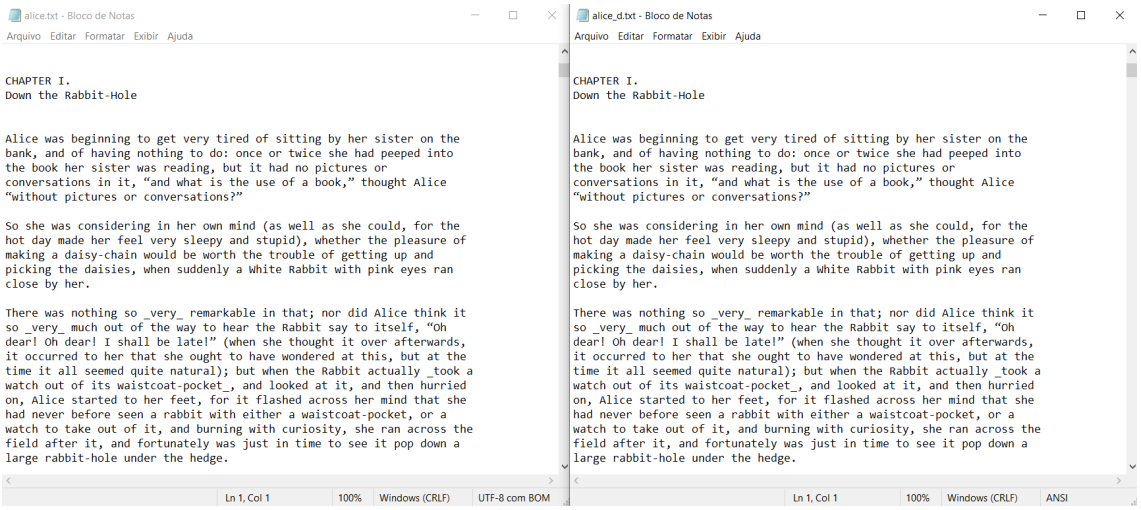

Figure 1: O arquivo original, a esquerda. Após a descompressão, a direita

2.4 Execução

Para execução do programa, a chamada do arquivo via linha de comando se dá semelhante ao solicitado pela descrição do trabalho prático. Para a compressão e descompressão deve ser feito, respectivamente:

- python main.py -c nome_do_arquivo_entrada.txt -o nome_do_arquivo_saída
- python main.py -x nome_do_arquivo_entrada.z78 -o nome_do_arquivo_saída

Para a compressão é gerado um arquivo de extensão *.z78*. Note é preciso declarar a extensão do arquivo de entrada, contudo para o arquivo de saída basta inserir o nome desejado (ausente da extensão).

3 Conclusão

Por fim conclui-se que a compressão do algoritmo opera bem quando a conversão do tipo utiliza de poucos bytes no arquivo de saída para representação. A Trie como dicionário para a compressão também se mostrou bastante eficiente, retornando em tempo hábil os índices das palavras requeridas, o qual foi a sua principal função neste trabalho prático. Por fim, a descompressão funcionou incrivelmente bem, sem muita perda de informação, indicando que o algoritmo de compressão de fato funciona como esperado. O saldo geral do trabalho prático é considerado positivo, já que reforçou o aprendizado das estruturas vistas em aula e pode-se trabalhar com aplicações práticas da manipulação de sequências e reconhecimento de padrões

4 Referências

- MULTIMEDIA.UFP.PT. Codificação baseada em dicionários: LZ78. Disponível em: http://multimedia.ufp.pt/codecs/compressao-sem-perdas/codificacao-baseada-em-dicionarios/lz78/. Acesso em: 09 maio 2023.
- WIKIPÉDIA, a enciclopédia livre. LZ78. Disponível em: https://pt.wikipedia.org/wiki/LZ78. Acesso em: 09 maio 2023.