Trabalho Prático 1
Manipulação de Sequências

Jean George Alves Evangelista - 2018047021

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil

jeanalves@dcc.ufmg.br

1. Introdução

O objetivo deste trabalho é a realizar a implementação de um programa que realize a compressão e a descompressão de arquivos utilizando o algoritmo LZ78. O LZ78 trabalha substituindo *strings* que se repetem no texto por um código e dessa maneira diminui o número de bytes gravados na saída. O algoritmo utiliza um dicionário e esse obrigatoriamente deve ser implementado por meio de uma trie compacta. A implementação e os testes realizados estão explicados nas seções a seguir.

2. Implementação

Optou-se por realizar a implementação na linguagem de programação Python (versão 3.9), por ser uma linguagem de mais alto nível se comparada à C/C++, fornecendo assim maneiras muitos mais simples para se trabalhar com strings, arquivos, etc. O programa foi subdividido em quatro módulos (arquivos com extensão .py): *utils*, *main*, *node* e *trie*.

2.1 Utils

Um módulo pequeno que conta com alguns métodos comuns. Tais métodos poderiam ser adicionados no arquivo *main.py* mas por organização optou-se por deixá-los em *utils.py*. Não há muito o que comentar por serem métodos bem simples.

2.2 Main

Primeiro módulo a ser chamado no programa, aqui são chamados os métodos de compressão/descompressão e são realizados os testes.

A execução é iniciada no método *main*. É feita uma validação na quantidade de argumentos recebidos e o programa tenta receber os seguintes parâmetros:

- tipo da operação: “-c” para compressão, “-x” para descompressão e -t para testes. “-t” não foi especificado mas foi adicionado para facilitar nos testes (mais detalhes na seção de testes). Caso não seja “-c”, “-x” ou “-t” o programa se encerra.

- arquivo de entrada: É obrigatório e é validado se sua extensão é “.z78” ou “.txt”. Essa validação de extensão não foi especificada mas foi entendido que o programa trabalhará apenas com essas duas extensões, para trabalhar com outros formatos seria necessário adaptar a implementação.
- arquivo de saída: É opcional, indica qual será o nome do arquivo de saída para a compressão/descompressão. Caso seja recebido é feita a mesma validação do arquivo de entrada.

A partir daqui, o método *main* chama a compressão, descompressão ou função de testes de acordo com o tipo da operação recebido.

O método de compressão abre o arquivo de texto, pega seu conteúdo, passa para um método da classe *Trie* e espera receber o texto comprimido. Por fim, tenta escrever no arquivo binário de saída. A implementação da compressão é realizada dentro da classe *Trie* presente no módulo *trie.py* e por isso será explicada na respectiva seção.

O método de descompressão abre o arquivo binário e lê seu conteúdo. Em seguida, o conteúdo é percorrido de maneira a se obter o *index* utilizado no algoritmo de compressão e o caractere referente a esse *index*. De acordo com a especificação do algoritmo, o texto após ser comprimido possui sempre o formato Index1Caractere1Index2Caractere2 e assim por diante. Por exemplo, a *string* “A_ASA_DA_CASA” seria comprimida para “0A0_1S1_0D4C3A”. Como os caracteres em UTF-8 podem ser representados em diferentes quantidades de *bytes* foi necessário realizar uma verificação para saber se o caractere ocupa 1, 2, 3 ou 4 bytes e dessa maneira pegar o próximo *index* corretamente. Essa foi das partes mais difíceis da implementação e antes de realizá-la caracteres especiais não estavam sendo formatados corretamente. Para realizar a descompressão foi utilizada uma estrutura simples de dicionário, por não existir necessidade de trabalhar com a trie, que foi dada como obrigatória apenas para a compressão. Após percorrer os bytes e converter para texto, o arquivo de saída é aberto e o texto é salvo nele. É feita uma verificação no texto antes que ele seja salvo, pois foi adicionado um "caractere especial” para evitar casos em que o último caractere do texto era ignorado.

O método de testes será explicado na seção de testes.

2.3 Node

O módulo *node.py* define a classe *Node*, que é um nó da trie implementada. Um nó armazena um caractere, o índice da última palavra inserida, uma *flag* que diz se ele é o fim de uma palavra e um dicionário para armazenar nós filhos.

2.3 Trie

De acordo com a especificação era necessário implementar o dicionário utilizado no LZ78 por meio de uma Trie compacta. Entretanto, compactar a trie da maneira explicada em aula não se mostrou uma tarefa simples e não foi possível implementar a

compactação em tempo hábil para a entrega. Ao invés disso, foi utilizada uma trie comum, onde cada nó armazena apenas um caractere. A compressão/descompressão funcionou corretamente para a trie.

A trie possui um nó raiz, um atributo que diz o índice do nó inserido na trie (bastante útil no momento da compressão), atributos indicativos do tipo de decodificação utilizado (no caso UTF-8) e da quantidade de bytes utilizados para um número (no caso 3) e um *array* de *bytes* que representa o texto após comprimido.

O método principal do programa é o *populate_and_compress*, que recebe por parâmetro um texto e o insere na trie, atualizando sempre o atributo que armazena o texto comprimido. O que esse método faz é exatamente executar o algoritmo LZ78:

- percorre o texto recebido caractere a caractere
- verifica se uma palavra está na trie
    - se não estiver, insere e atualiza o texto comprimido
    - se estiver e for o final do texto, adiciona um "caractere especial" no fim do texto comprimido. Isso foi necessário porque em alguns casos a descompressão ignorava o último caractere do texto.

O método de inserção na trie sempre tenta inserir a partir da raiz, percorrendo a palavra recebida caractere a caractere e buscando o filho correspondente. Caso o filho não esteja presente ele é criado. No fim, teremos sempre todos os caracteres da palavra na trie e atualizamos o contador do índice e o indicador de fim de palavra do nó.

O método de busca também percorre a trie a partir da raiz e a palavra recebida caractere a caractere. O que ele faz basicamente é verificar sempre se existe um filho com determinado caractere, se não existir retorna *false*. No fim, caso saia do *loop* teremos duas possibilidades:

- a palavra recebida existe na trie, mas não é uma palavra efetivamente porque não há indicação de fim de palavra no último nó. Nesse caso, retornamos que a palavra não está presente na trie.
- a palavra recebida existe na trie e existe indicação de fim de palavra no último nó, nesse caso retornamos *true* e também o índice daquele nó, visto que o algoritmo de compressão (que é quem chama esse método) precisa dele.

3. Testes

Segundo a especificação, era necessário adicionar no mínimo 10 arquivos reais como exemplos e calcular a taxa de compressão para cada um. Os exemplos escolhidos foram livros famosos retirados do site *Project Gutenberg,* em formato *.txt*. Os livros escolhidos foram:

1. "*Alice's Adventures in Wonderland*", de Lewis Carroll - 73 kB
2. "*Dracula*", de Bram Stoker - 862 kB
3. "*Frankenstein*", de Mary Wollstonecraft Shelley - 438 kB
4. "*Leviathan*", de Thomas Hobbes - 1.2MB

5. “*Metamorphosis*”, de Franz Kafka - 138 kB
6. “*Moby Dick*”, de Herman Melville - 1.2 MB
7. “*Persuasion*”, de Jane Austen - 483 kB
8. “Peter *Pan*”, de J. M. Barrie - 284 kB
9. “*The Iliad*”, de Homero - 1.1 MB
10. “*The Republic*”, de Platão - 1.2MB

Além dos 10 textos acima, foram adicionados outros:

- 2mb.txt: Um arquivo no padrão *Lorem ipsum,* adicionado exclusivamente para testar o tamanho máximo especificado. Tamanho: 2 MB
- emoji.txt: Um arquivo que mistura texto e emojis, adicionado para testar caracteres especiais. Tamanho: 2.6 kB
- korean.txt: Um texto em coreano adicionado para testar caracteres em linguagens diferentes. Tamanho: 1.5 kB
- tp1.txt: A própria especificação deste trabalho prático, adicionado para testar um texto em português. Tamanho: 5.4kB

Os arquivos fonte dos textos acima podem ser encontrados na pasta *tests* dentro do projeto. Para trabalhar com os textos basta adicionar o caminho correto no momento de executar o programa, ex: “python main.py -c **tests/leviathan.txt**”.

Entretanto, para facilitar nos testes o programa foi adaptado para lidar com a operação “-t”, que indica que estamos querendo executar os testes. A execução do programa nesse caso deve ser a seguinte: “python -t nome_do_arquivo.txt”, onde “nome_do_arquivo.txt” indica o nome do arquivo onde salvaremos as informações obtidas nos testes. Observação: o arquivo será criado na pasta */tests/results*.

O método *tests* criado utiliza métodos nativos da linguagem para abrir todos os arquivos de texto presentes na pasta *tests* e para cada um é feita sua compressão. Em seguida, é realizada a descompressão do arquivo feito. O arquivo comprimido e descomprimido são salvos e a taxa de compressão é calculada com base nos tamanhos dos arquivos. O arquivo original também é comparado com sua cópia, afim de saber se o tudo está implementado corretamente. No fim, os resultados são salvos no arquivo de saída, de acordo com o nome recebido.

Foram realizados testes para todos os arquivos listados mais acima e em todos os casos o arquivo original esteve igual ao que passou pelo processo de compressão -> descompressão, portanto o programa funcionou corretamente. As taxas de compressão são apresentadas na tabela abaixo.

Arquivo	Taxa de Compressão
alice.txt	1.30
dracula.txt	1.50
frankenstein.txt	1.42
leviathan.txt	1.66
metamorphosis.txt	1.27
moby_dick.txt	1.53
persuasion.txt	1.47
peter_pan.txt	1.27
the_iliad.txt	1.53
the_republic.txt	1.65
2mb.txt	2.42
emoji.txt	0.99
korean.txt	0.98
tp1.txt	0.80

Tabela 1. Resultado de taxa de compressão obtidos nos testes

A taxa de compressão é dada pelo tamanho do arquivo original dividido pelo tamanho do arquivo comprimido. Sendo assim, é desejável que ela seja maior do que um, o que indica que o arquivo comprimido é menor que o original. Além disso, valores altos de taxa de compressão indicam que o arquivo comprimido é consideravelmente menor do que o arquivo original. Na tabela de resultados acima notamos que para quase todos os casos foram obtidos uma taxa de compressão maior do que um, com destaque para o arquivo *2mb.txt*, que é um arquivo que repete muitas sequências de caracteres e algoritmo reduz bastante seu tamanho. Já para arquivos menores, como os três últimos, a taxa de compressão se mostrou ligeiramente menor do que um, um indicativo de que o arquivo comprimido é maior do que o original. Isso aconteceu provavelmente porque estamos sempre inserindo um caractere no fim do texto ao comprimi-lo, o que pode acabar impactando nos valores de armazenamento para arquivos menores.

4. Instruções de execução

A execução das funcionalidades de compressão e descompressão se dá conforme especificado, sendo necessário executar o arquivo *main.py*. Exemplos:

python main.py -c tests/leviathan.txt

python main.py -x tests/leviathan.z78 tests/leviathan.txt

Para executar os testes implementados dentro do programa basta executar o comando:

python main.py -t results.txt.

Será criado um arquivo results.txt dentro da pasta tests/results.

5. Conclusão

Mesmo sem realizar a implementação da trie compactada é possível concluir que o programa de compressão/descompressão utilizando o LZ78 foi implementado com sucesso. O desenvolvimento do trabalho foi bastante interessante por permitir trabalhar com a linguagem de programação Python, uma estrutura de dados mais avançada (trie) e um método de compressão de arquivos. O LZ78, embora não muito avançado, é base para outros e ajuda a entender bem como computadores realizam compressões. Além disso, foi possível exercitar na prática alguns dos conceitos relacionados à manipulação de sequências vistos em sala de aula.

Referências

Python 3.9.1 documentation. Disponível em: <https://docs.python.org/3/>. Acesso em: 07 fev. 2021.

StackOverflow. Disponível em: <https://stackoverflow.com/>.Acesso em: 07 fev. 2021.

LZ78. Disponível em: <https://pt.wikipedia.org/wiki/LZ78>. Acesso em: 07 fev. 2021.

David Salomon. Data Compression: The complete reference. 4th edition. Springer